Componentes Reutilizáveis em Java com

Reflexão e Anotações

EDUARDO GUERRA

Sumário

Versão: 16.9.7

Parte I

Conceitos Básicos

Essa primeira parte do livro vai ensinar os conceitos a respeito de reflexão e metadados assim como o uso das APIs da linguagem Java para a utilização desses recursos. A partir da leitura desses quatro primeiros capítulos, espera-se que o leitor obtenha familiaridade com o uso de reflexão na linguagem Java, sabendo como obter e manipular seus principais elementos, utilizando-os para desenvolver componentes reutilizáveis em diferentes contextos.

O capítulo 1 fala sobre as deficiências da orientação a objetos e faz uma introdução a reflexão computacional, mostrando seu poder através de um pequeno exemplo. Já o capítulo 2 entra em detalhes a respeito da API Reflection da linguagem Java, mostrando seus principais elementos e como podem ser utilizados. Em seguida, o capítulo 3 aborda a configuração de metadados adicionais, falando sobre as diferentes alternativas para defini-los e entrando em detalhes sobre como as anotações funcionam na linguagem Java. Finalizando essa primeira parte, o capítulo 4 fala sobre o recurso de proxies dinâmicos, quais as formas de implementá-los e como combiná-los com as técnicas mostradas nos capítulos anteriores. No final de cada capítulo é apresentado um exemplo realista que aplica os conceitos apresentados.

Capítulo 1

Conhecendo a Reflexão

"O mundo é como um espelho que devolve a cada pessoa o reflexo de seus próprios pensamentos."
– Luís Fernando Veríssimo

Me lembro até hoje quando estava começando a amadurecer nos conceitos da orientação a objetos. Um fato que me marcou muito foi quando pela primeira vez eu vi a necessidade de criar uma interface para encapsular um comportamento. Foi como se um novo mundo se abrisse na minha frente. Como diz a expressão em linguagem popular, "a ficha caiu". Eu finalmente compreendia como utilizar aquelas funcionalidades da orientação a objetos para implementar de uma forma inteligente os requisitos de um sistema. Com o estudo dos padrões de projeto, esse conhecimento foi cada vez mais se refinando, até ser concretizado no livro "Design Patterns com Java: Projeto orientado a objetos guiado por padrões". Só que a minha jornada para desenvolver software de uma forma cada vez melhor não parou na orientação a objetos...

Com o tempo, eu fui vendo que apenas com a utilização da orientação a objetos

muitos problemas ainda não eram resolvidos. Foi aí que conheci a reflexão e um novo mundo se abriu novamente a minha frente! A partir da reflexão é possível, em tempo de execução, conhecer a estrutura de uma classe e utilizar essas informações para a execução de uma lógica. Dessa forma, alguns tipos de código que dependiam de cada classe, mas que eram repetitivos em sua essência, agora podem ser criados de forma mais geral e reutilizados em diversos contextos.

A partir desse conhecimento de reflexão e das ferramentas para aplicar isso na linguagem Java, comecei a desenvolver soluções reutilizáveis no meu dia-a-dia, enxergando o potencial dessa técnica em aumentar a reutilização de código a partir da eliminação de tarefas repetitivas. De forma complementar, comecei a utilizar metadados, a princípio apenas com anotações de código, e em seguida combinando essas informações com fontes externas e padrões de nomenclatura. Como pouco estudo sobre esse tipo de solução havia sido conduzido até o momento, decidi assumir esse desafio em meus estudos de doutorado. Comecei então a estudar o código de diversos frameworks utilizados no mercado e abstraindo suas soluções consegui enxergar diversos padrões, tanto para sua estrutura interna, quanto para identificar cenários onde esse tipo de solução é aplicável.

Sendo assim, o objetivo desse livro é apresentar passo a passo todo o conhecimento que adquiri nessa jornada! Desde os primeiros passos no conhecimento da reflexão até técnicas avançadas para a utilização dos metadados. O que esse livro se diferencia de outros que ensinam a utilizar reflexão em Java, é que ele não se limita a mostrar como funcionam as APIs, mas também apresentam como elas podem ser utilizadas no projeto de um software para aumentar a reutilização e, consequentemente, a produtividade da equipe. Em outras palavras, será mostrado não só como usar, mas como utilizá-la da forma correta e mais eficiente!

Esse capítulo explora alguns problemas que a orientação a objetos não consegue resolver de forma direta, e introduz o conceito de reflexão, mostrando como ela funciona, especialmente na linguagem Java. Para que os leitores possam começar a saborear o poder da reflexão, alguns exemplos de código irão mostrar um pouco do que é possível fazer com reflexão. Em seguida, esse capítulo mostra que tudo tem um preço, e faz uma análise comparativa de desempenho na invocação de um método via reflexão com sua chamada direta. Para finalizar, é feita uma apresentação de como o livro está organizado, fazendo uma prévia do que será apresentado em cada capítulo.

1.1 Ainda falta alguma coisa na orientação a objetos?

A orientação a objetos possui diversos recursos poderosos, que nos permite modelar um sistema de forma a aumentar o reuso, permitindo a criação de um código com melhor qualidade. A utilização de padrões potencializa ainda mais a utilização desses recursos, pois são soluções recorrentes que possuem uma estrutura adequada para um conjunto de cenários. Porém, mesmo assim, ainda existem cenários em que a orientação a objetos não ajuda muito.

Lendo parâmetros de aplicações Web

Um exemplo comum desse tipo de código é a recuperação das informações de uma requisição web para serem inseridas em um outro objeto. O trecho de código a seguir ilustra essa situação com o código de dois servlets, sendo que um deles recebe informações referentes a um produto e o outro referente a um cliente. Se observarmos os dois códigos, eles são bem parecidos, porém apenas utilizando as técnicas de orientação a objetos, não é possível a reutilização.

Listagem 1.1 - Códigos de servlets para popular propriedades de objetos:

```
// código em um servlet que recupera informação de um produto

@WebServlet("/novoProduto")
public class NovoProdutoServlet extends HttpServlet{
    protected void doPost(HttpServletRequest rq, HttpServletResponse rp)
            throws ServletException, IOException {
        Produto p = new Produto();
        p.setNome(rq.getParameter("nome"));
        p.setCategoria(rq.getParameter("categoria"));
        p.setPreco(Double.parseDouble(rq.getParameter("preco"));
        p.setDescricao(rq.getParameter("descricao"))
        //outras informações
    }
}

// código em um servlet que popula informações de um novo cliente

@WebServlet("/cadastro")
public class CadastroClienteServlet extends HttpServlet{
```

```
    protected void doPost(HttpServletRequest rq, HttpServletResponse rp)
            throws ServletException, IOException {
        Cliente c = new Cliente()
        c.setNome(rq.getParameter("nome"));
        DateFormat formatadorData = new SimpleDateFormat("MM/dd/yy");
        c.setDataNascimento(
            (Date)formatter.parse(rq.getParameter("dataNascimento"));
        c.setLogin(rq.getParameter("login"));
        c.setSenha(rq.getParameter("senha"));
        //outras informações
    }
}
```

Um fato interessante de ser notado é que ambos os códigos seguem uma mesma lógica, ou seja, nos dois códigos parâmetros da requisição web são recuperados e inseridos no objeto em uma propriedade com o mesmo nome. Existem alguns casos especiais, como a conversão para `double` do preço do produto e a conversão para `Date` da data de nascimento do cliente. Mas, de qualquer forma, é fácil de notar a similaridade entre os códigos. Esse tipo de código normalmente é massante de ser criado, principalmente quando o número de parâmetros a serem recuperados é grande. Como consequência, frequentemente são cometidos erros onde uma `String` acaba sendo escrita errada ou uma das propriedades acaba sendo esquecida pelo desenvolvedor.

Uma outra questão é que se por acaso for adicionado mais um atributo em alguma das classes, isso vai acarretar em uma mudança de código no Servlet, onde um novo parâmetro vai precisar ser lido e atribuído a ele. Essa necessidade de mudança é um sintoma de que esse código poderia ser melhorado ou simplificado de alguma forma.

Criando proxies para executar métodos de forma assíncrona

Vamos agora considerar um exemplo bem diferente do anterior. Imagine que um sistema possua um serviço de logging, ou seja, para o registro de informações de auditoria, que possua métodos para registrar informações e erros. O contrato desse serviço é representado pela interface `LoggingService`. Nesse sistema, existem várias implementações desse serviço, que podem, por exemplo, armazenar as informações em banco de dados, em arquivos e etc... Devido a questões de desempenho, decidiu-se que esse serviço deveria executar de forma paralela a funcionalidade principal.

Seguindo um bom design orientado a objetos, e principalmente por existirem diversas implementações dessa interface, decidiu-se fazer um *proxy* que encapsula a classe original e executa seus métodos em uma nova *thread*. Segue abaixo como seria o código dessa classe. Ela implementa a interface `LoggingService` e possui um atributo desse mesmo tipo, que irá conter o objeto encapsulado pelo *proxy*. Observe que a cada chamada de método, o mesmo método do objeto encapsulado é invocado em uma thread diferente.

Listagem 1.2 - Proxy para executar as funções de logging de forma assíncrona:

```
public class LoggingAsyncProxy implements LoggingService{

    private final LoggingService service;

    public LoggingAsyncProxy(LoggingService service){
        this.service = service;
    }
    public void registerInformation(String info){
        new Thread(
            public void run(){
                service.registerInformation(info);
            }
        ).start();
    }
    public void registerError(Exception error){
        new Thread(
            public void run(){
                service.registerError(error);
            }
        ).start();
    }
}
```

Se observar o código do *proxy* criado, é possível perceber que existe uma certa duplicação nos métodos, porém essa é uma duplicação difícil de ser removida pois o método que é invocado é diferente. Imagine agora que seja necessário criar essa mesma funcionalidade para uma classe que possui uma interface diferente. Imagine, por exemplo, em uma classe do tipo DAO, *Data Access Object*, responsável por persistir e recuperar objetos, que se deseje executar uma *thread* diferente os métodos que não tenham retorno. O código resultante do código criado está mostrado a

seguir.

Listagem 1.3 - Proxy para executar as funções de logging de forma assíncrona:

```
public class ProdutoDAOAsync implements ProdutoDAO{

    private final ProdutoDAO dao;

    public ProdutoDAOAsync(ProdutoDAO dao){
        this.dao = dao;
    }
    public void inserir(Produto p){
        new Thread(
            public void run(){
                dao.inserir(p);
            }
        ).start();
    }
    public Produto recuperar(int id){
        return dao.recuperar(id);
    }
    public void excluir(Produto p){
        new Thread(
            public void run(){
                dao.excluir(p);
            }
        ).start();
    }
    public void atualizar(Produto p){
        new Thread(
            public void run(){
                dao.atualizar(p);
            }
        ).start();
    }
}
```

Se compararmos, é possível observar que os dois códigos são muito parecidos, porém podemos também ver que é difícil de reaproveitar o código de um no outro caso. O motivo dessa dificuldade é que por mais que a funcionalidade seja parecida, ela precisa invocar um método da classe que está sendo encapsulada no meio de sua

execução. Dessa forma, é até possível reutilizar essa mesma classe de proxy para classes que implementem uma mesma interface, mas para interfaces diferentes fica mais complicado.

1.2 Reflexão, muito prazer!

"Vá para o seu negócio, o prazer, enquanto eu vou para meu prazer, negócios."
– William Wycherley

A noção de reflexão computacional foi introduzida pela primeira vez, ainda no contexto de linguagens procedurais, em 1982 [5]. Porém foi em 1987, que o artigo *"Concepts and Experiments in Computational Reflection"* [3] consolidou o conceito. Reflexão pode ser definida como o processo no qual um programa de computador pode observar e modificar sua própria estrutura e comportamento. A utilização de reflexão também é conhecida como metaprogramação, pois com sua utilização um programa pode realizar computações a respeito dele próprio.

A introspecção é um subconjunto da reflexão que permite um programa a obter informações a respeito de si próprio. A partir das informações obtidas com a introspecção, é possível manipular instâncias acessando seus atributos e invocando seus métodos. Dessa forma, é possível criar criar um código que lida com uma classe cuja estrutura ele não conhece.

A reflexão também já foi documentada como um padrão no primeiro livro da série "Pattern-Oriented Software Architecture", conhecido informalmente como POSA [2]. Segundo essa fonte, um dos benefícios de sua utilização está na flexibilidade e adaptabilidade do código. Se um código que utiliza reflexão se baseia na estrutura de uma classe, por exemplo, ele se adaptará mais facilmente a mudanças nessa estrutura. Ainda segundo esse padrão, a piora no desempenho é uma das desvantagens de seu uso, assim como um aumento na complexidade do código.

Todos os programadores sabem que para executar um programa, um computador deve carregar seus comandos na memória. Adicionalmente, ele também utiliza a memória para armazenar variáveis que são manipuladas como parte da execução do software. Dentro desse contexto, a reflexão nada mais é do que o programa acessar e modificar as suas instruções. Esse é um recurso poderoso, mas que precisa ser utilizado com bastante responsabilidade.

Reflexão em Java

Em Java, as classes que implementam funcionalidades relacionadas a reflexão ficam no pacote `java.lang.reflect` e são conhecidas como **API Reflection**. Apesar do nome, grande parte dessas classes na verdade implementam apenas funções de introspecção. Em outras palavras, é possível obter informações sobre as classes de um software, mas não é possível modificá-las. Grante parte desse livro foca principalmente nessas funcionalidades.

A alteração de classes em Java não é suportada pela API padrão de reflexão, mas pode ser feita através da manipulação de bytecode. De uma forma mais simples, a aplicação modifica o código compilado que será carregado pela máquina virtual. Isso pode ser feito estaticamente depois da compilação do código, no momento do carregamento da classe ou em tempo de execução. Como é complicado substituir na máquina virtual uma classe que já foi carregada, quando a modificação é feita em tempo de execução, normalmente é carregada uma nova classe com as modificações feitas na classe existente. O penúltimo capítulo desse livro é dedicado a técnica de manipulação de bytecode, porém em capítulos anteriores são apresentadas algumas ferramentas que utilizam esse recurso de uma forma encapsulada.

Diferença em Relação a Linguagens Dinâmicas

O termo **linguagem dinâmica** é utilizado para descrever linguagens de alto nível que executam em tempo de execução comportamentos que as linguagens normalmente fazem durante a compilação. Dentre esses comportamentos normalmente está a adição de código em objetos e a modificação de tipos. Nesse tipo de linguagem esse tipo de operação é comum e muitas vezes é suportada a partir de recursos e ferramentas da própria linguagem. Exemplos desse tipo de linguagem são Ruby, Python e Groovy.

A utilização de reflexão em linguagens dinâmicas acaba sendo mais simples do que em linguagens estaticamente tipadas como Java. Em alguns casos, a invocação dinâmica de métodos e a modificação de tipos é utilizada como mecanismo padrão da linguagem, e não apenas em situações que se deseja uma maior flexibilidade. As técnicas mostradas nesse livro focam em como utilizar com segurança os recursos da reflexão em Java, que é uma linguagem estaticamente tipada. Dessa forma, a aplicação pode usufruir de recursos que permitem uma maior segurança de código e aplicar a reflexão somente em pontos onde uma maior flexibilidade é necessária. Por mais que o uso desses recursos seja diferente em linguagens dinâmicas, acredito que alguns conceitos aqui apresentados também podem ser utilizados nesse contexto.

Como a reflexão pode me ajudar?

A utilização da reflexão é muito útil em situações em que o código precisa lidar com classes com interfaces diferentes, porém sua criação é repetitiva e segue uma lógica. Para esse tipo de código acaba sendo difícil a utilização de técnicas tradicionais da orientação a objetos, pois os objetos possuem contratos diferentes e os métodos que precisam ser invocados dependem dos objetos. Os exemplos apresentados no começo desse capítulo ilustram dois cenários onde isso ocorre.

A ideia do uso de reflexão é tornar o código adaptável a estrutura dos objetos e com isso permitir que ele possa ser mais reutilizável. Dessa forma, é possível substituir a criação de um código que precisaria ser repetido para cada classe, pela chamada a um método ou uma classe que utilize reflexão. Dessa forma é possível diminuir a

quantidade de código, aumentando a velocidade de desenvolvimento da equipe.

Por mais que a utilização de reflexão seja uma técnica mais complexa, seu uso pode ser feito de forma bem localizada. Em outras palavras, é possível encapsular sua utilização dentro de classes ou componentes, de forma que isso fique transparente para o resto do software. Um grande exemplo disso são os frameworks, pois grande parte deles utilizam esses recursos de reflexão de forma encapsulada e isso não precisa ser conhecido pelo desenvolvedor que está o utilizando.

1.3 O primeiro contato com a API Reflection

Como também sou programador, sei que grande parte dos leitores devem estar ansiosos para ver um pouco de código com a reflexão sendo utilizada na prática. Essa seção apresenta uma pequena prévia sobre o funcionamento da API de reflexão em Java, mostrando um pequeno exemplo que ilustra a sua utilização. Aqui também falaremos das anotações e como podem ser utilizadas em conjunto com a reflexão. Peço para que nesse momento se preocupem apenas em entender os conceitos e o exemplo, pois maiores detalhes sobre essas APIs serão dados nos próximos capítulos.

Transformando as propriedades de uma classe em mapas

O exemplo que vamos explorar nesse primeiro capítulo envolve a geração de um mapa de propriedades a partir de uma classe no formato Java Bean, ou seja, com métodos de acesso no formato get e set. Ao criar um algoritmo utilizando reflexão, a ideia é ver qual seria o procedimento que um desenvolvedor utilizaria para escrever o código manualmente e tentar reproduzir esse mesmo procedimento no código. No caso da geração do mapa, o procedimento é basicamente invocar cada método getter da classe e adicionar no mapa o valor retornado com a chave do nome da propriedade.

Esse exemplo, de certa forma, é parecido com o que vimos anteriormente para a recuperação dos parâmetros do Servlet. No exemplo anterior precisávamos recuperar os parâmetros passados para página de forma a popular as propriedades do objeto e aqui iremos recuperar as propriedades de um objeto para inserirmos em um mapa. Em ambos os casos, caso fosse desenvolvido um código para cada classe, esse código seria repetitivo e chato de ser criado. O objetivo é automatizar esse tipo de tarefa, de forma que esse tipo de código não precise ser criado, diminuindo a chance de erros e aumentando a produtividade dos desenvolvedores.

A classe apresentada a seguir mostra uma possível implementação desse código

para gerar o mapa de propriedades. O método `gerarMapa()` percorre todos os métodos do objeto passado como parâmetro, recuperando como retorno somente os métodos identificados como getters e inserindo no mapa. Para poder acessar os dados de uma classe, o primeiro passo é recuperar a instância da classe `Class` relativa a classe do objeto, o que é feito através do método `getClass()`. A partir dessa instância, os métodos são recuperados através de `getMethods()` e essa lista de instâncias de `Method` é percorrida procurando os que podem ser classificados como getter.

O método `isGetter()` acessa as informações do método verificando por suas informações se se trata de um método getter. As informações consideradas para isso são: se nome do método se inicia com "get", se o retorno é diferente de `void` e se ele não possui parâmetros. Caso o método seja identificado como getter, ele é invocado a partir do método `invoke()` e seu valor recuperado para ser inserido no mapa. Observe que o nome do método é transformado no nome da propriedade pelo método `deGetterParaPropriedade()` para adição no mapa. Outra questão interessante de ser notada é que a invocação do método via reflexão está envolta por um bloco `try/catch`, o que indica que existem erros que podem ocorrer nessa invocação (o que será abordado mais a frente nesse livro).

Listagem 1.4 - Classe que utiliza reflexão para a geração de um mapa:

```
public class GeradorMapa {
    public static Map<String, Object> gerarMapa(Object o){
        Class<?> classe = o.getClass();
        Map<String, Object> mapa = new HashMap<>();
        for(Method m: classe.getMethods()){
            try {
                if(isGetter(m)){
                    String propriedade = deGetterParaPropriedade(
                            m.getName());
                    Object valor = m.invoke(o);
                    mapa.put(propriedade, valor);
                }
            } catch (Exception e) {
                throw new RuntimeException(
                    "Problema ao gerar o mapa",e);
            }
        }
        return mapa;
```

```
    }
    private static boolean isGetter(Method m) {
        return m.getName().startsWith("get") &&
            m.getReturnType() != void.class &&
            m.getParameterTypes().length == 0;
    }
    private static String deGetterParaPropriedade(String nomeGetter){
        StringBuffer retorno = new StringBuffer();
        retorno.append(nomeGetter.substring(3, 4).toLowerCase());
        retorno.append(nomeGetter.substring(4));
        return retorno.toString();
    }
}
```

A próxima listagem mostra a utilização desse método criado para a geração de mapas. A classe `Produto`, a mesma do exemplo do Servlet, possui quatro propriedades com métodos get e set para sua respectiva recuperação e modificação. Em seguida é apresentado um código que cria uma nova instância dessa classe, recupera o mapa com o método criado e imprime as propriedades recuperadas no console. Sendo assim, é possível ver que com reflexão foi possível criar uma rotina que acessa métodos de uma classe sem depender diretamente dela.

Listagem 1.5 - Utilização do método de geração do mapa de propriedades:

```
// Classe para ser utilizada como base
public class Produto {

    private String nome;
    private String categoria;
    private Double preco;
    private String descricao;

    public Produto(String nome, String categoria, Double preco,
                    String descricao) {
        this.nome = nome;
        this.categoria = categoria;
        this.preco = preco;
        this.descricao = descricao;
    }
    //metodos get e set omitidos
}
```

```
//Código que executa o método de geração do mapa e imprime
public static void main(String[] args){

    Produto p = new Produto("Design Patterns","LIVRO",59.90,
            "Publicado pela Casa do Codigo");
    Map<String,Object> props = GeradorMapa.gerarMapa(p);
    for(String prop : props.keySet()){
        System.out.println(prop+" = "+props.get(prop));
    }
}
```

Conhecendo anotações

Quando você implementa um método para cada classe, por mais que ele seja repetitivo, tem-se a liberdade de realizar alterações para atender necessidades de negócio. Tomando como exemplo a geração do mapa, poderia ser necessário que alguma propriedade precisasse ser ignorada ou que uma propriedade fosse adicionada com um nome diferente. Ao criar cada método individual isso é fácil de fazer, porém, utilizando reflexão, como fazer para que o algoritmo trate algumas propriedades diferentes? Apesar da reflexão permitir a definição de um método genérico para a geração de mapas, ele não atende os requisitos quando algum elemento precisa ser tratado de forma diferente.

É nesse ponto que entram as anotações! Elas permitem marcar os elementos de forma que um algoritmo que utilize reflexão possa identificar os elementos que ele deve tratar de forma diferente. As anotações permitem adicionar novas informações em elementos de programação. Essas informações são chamadas de específicas de domínio, pois elas são dizem respeito ao interesse da classe que irá consumi-las. É importante já deixar claro que uma anotação não possui comportamento e somente sua presença não faz nada além de adicionar uma informação, porém elas podem ser acessadas por outras classes e componentes para permitir que eles adaptem seu comportamento de acordo com sua presença.

As duas listagens a seguir mostram como as anotações poderiam ser criadas, sendo a primeira para ignorar uma propriedade e a segunda para definir um nome diferente para uma propriedade. Novamente peço para os leitores não se preocuparem muito com os detalhes de implementação, que serão explicados nos próximos capítulos. O que é importante perceber nessas anotações é que a primeira não possui propriedades e a segunda possui uma propriedade do tipo `String`

chamada `value`.

Listagem 1.6 - Anotação que define quando uma propriedade precisa ser ignorada:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignorar {
}
```

Listagem 1.7 - Anotação que define um nome diferente para a propriedade:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface NomePropriedade {
    String value();
}
```

Como já foi dito anteriormente, nada adianta criar anotações sem um código que as consome, pois sozinhas elas não fazem nada. Dessa forma, a próxima listagem mostra como o código do gerador de mapas poderia ser modificado para considerar a presença das anotações. No método `isGetter()` foi acrescentada uma condição que verifica se a anotação `@Ignorar` está presente, fazendo com que os métodos getters com essa anotação não sejam considerados. Adicionalmente, no método principal `gerarMapa()`, antes de chamar o método `deGetterParaPropriedade()` é verificada se a anotação `@NomePropriedade` está presente e, em caso positivo, o valor configurado é recuperado e utilizado.

Listagem 1.8 - Modificação da classe que gera o mapa para considerar as anotações:

```
public static Map<String, Object> gerarMapa(Object o){
    Class<?> classe = o.getClass();
    Map<String, Object> mapa = new HashMap<>();
    for(Method m: classe.getMethods()){
        try {
            if(isGetter(m)){
                String propriedade = null;
                if(m.isAnnotationPresent(NomePropriedade.class)){
                    propriedade =
                        m.getAnnotation(NomePropriedade.class).value();
                }else{
```

```
                    propriedade = deGetterParaPropriedade(m.getName());
                }
                Object valor = m.invoke(o);
                mapa.put(propriedade, valor);
            }
        } catch (Exception e) {
            throw new RuntimeException("Problema ao gerar o mapa",e);
        }
    }
    return mapa;
}
private static boolean isGetter(Method m) {
    return m.getName().startsWith("get") &&
        m.getReturnType() != void.class &&
        m.getParameterTypes().length == 0 &&
        !m.isAnnotationPresent(Ignorar.class);
}
```

A partir dessa nova versão, é possível, a partir da adição de anotações, alterar parte do comportamento do algoritmo reflexivo, como fazendo-o ignorar uma propriedade ou mudando o nome que é adicionado no mapa. A seguir está um exemplo de como alguns dos métodos da classe `Telefone` seriam anotados para que o código do gerador de mapas conseguisse capturar as anotações. Mas esse exemplo foi só para dar um gostinho do que virá nos próximos capítulos do livro, onde será mostrado não só como utilizar reflexão e anotações, mas também em que situações e quais técnicas podem ser utilizadas.

Listagem 1.9 - Exemplo do uso das anotações pela classe Telefone:

```
public class Telefone {

    //atributos e outros métodos omitidos

    @NomePropriedade("codigoInternacional")
    public String getCodigoPais() {
        return codigoPais;
    }

    @Ignorar
    public String getOperadora() {
        return operadora;
```

```
    }
}
```

1.4 Usar reflexão tem um preço?

"There's no such thing as a free lunch. (Não existe esse negócio de almoço grátis.)"
– Expressão popular que expressa a ideia de que é impossível conseguir algo sem dar nada em troca

Um dos principais preços que se paga ao se utilizar reflexão é o desempenho. Invocar um método a partir de reflexão é mais demorado do que invocar um método diretamente no objeto. Isso é um fato! Porém somente essa informação não é suficiente para avaliar se isso pode gerar um problema de desempenho. Para termos noção do prejuízo ao desempenho trazido pela reflexão, essa seção apresenta um teste que compara o tempo de uma invocação de método feita diretamente com formas de invocar esse método utilizando a API de reflexão.

Teste para medida de desempenho

Para realizar esse teste, foi criada uma classe chamada `ClasseTeste` que possui um método chamado `metodoVazio()`. Como o próprio nome do método sugere, seu conteúdo é vazio para que o tempo medido seja apenas relativo a sua invocação. Em seguida, foi criada uma interface chamada `InvocadorMetodo`, que será o contrato de classes que irão invocar esse método repetidas vezes utilizando diferentes abordagens. O método `invocarMetodo()` recebe como parâmetro o número de vezes que o método precisa ser invocado e o chama repetidas vezes.

A primeira implementação dessa interface está representada na listagem a seguir. A classe `InvocadorNormal` faz a invocação do método diretamente através de uma instância da classe. Essa implementação servirá como base para compararmos com a invocação do método utilizando reflexão.

Listagem 1.10 - Implementação que invoca o método diretamente:

```
public class InvocadorNormal implements InvocadorMetodo {
    public void invocarMetodo(int vezes) {
        ClasseTeste ct = new ClasseTeste();
        for(int i=0; i<vezes; i++){
            ct.metodoVazio();
        }
```

```
    }
}
```

As duas listagens a seguir, `InvocadorReflexao` e `InvocadorReflexaoCache`, utilizam reflexão para invocar o método. Em ambas, a classe do objeto é recuperada através do método `getClass()` e o método recuperado pelo seu nome através de uma chamada a `getMethod()`. Em seguida, o método é invocado na instância chamando-se o método `invoke()`, passando-a como argumento. A diferença entre essas duas implementações é que na primeira a representação do método é recuperada a cada iteração e na segunda ela é recuperada uma vez e reaproveitada.

Listagem 1.11 - Implementação que a cada iteração busca o método e o invoca utilizando reflexão:

```
public class InvocadorReflexao implements InvocadorMetodo {
    public void invocarMetodo(int vezes) {
        try {
            ClasseTeste ct = new ClasseTeste();
            for(int i=0; i<vezes; i++){
                Method m = ct.getClass().getMethod("metodoVazio");
                m.invoke(ct);
            }
        } catch (Exception e) {
            throw
                new RuntimeException("Não consegui invocar o método",e);
        }
    }
}
```

Listagem 1.12 - Implementação que busca o método uma vez e o invoca utilizando reflexão a cada iteração:

```
public class InvocadorReflexaoCache implements InvocadorMetodo {
    public void invocarMetodo(int vezes) {
        try {
            ClasseTeste ct = new ClasseTeste();
            Method m = ct.getClass().getMethod("metodoVazio");
            for(int i=0; i<vezes; i++){
                m.invoke(ct);
            }
```

```
        } catch (Exception e) {
            throw
                new RuntimeException("Não consegui invocar o método",e);
        }
    }
}
```

Para comparar as formas de invocação das implementações de `InvocadorMetodo`, foi criada a classe `TesteDesempenho` apresentada a seguir. O método estático `executaTeste()` recebe como parâmetro uma instância de `InvocadorMetodo` para realizar o teste. Ele chama o método `invocarMetodo()` para que o método seja invocado 100.000 vezes, e mede o tempo que ele leva para executar em nanosegundos. Além de imprimir esse resultado no console, ele retorna o tempo medido.

Listagem 1.13 - Classe que compara as formas de invocação:

```
public class TesteDesempenho {
    public static void main(String[] args){
        double normal = executaTeste(new InvocadorNormal());
        double reflection = executaTeste(new InvocadorReflexao());
        System.out.println(
            (reflection/normal) + " vezes mais que o normal"
        );
        double reflectionCache = executaTeste(
            new InvocadorReflexaoCache());
        System.out.println(
            (reflectionCache/normal) + " vezes mais que o normal"
        );

    }
    public static double executaTeste(InvocadorMetodo invoc){
        long millis = System.nanoTime();
        invoc.invocarMetodo(100000);
        String nomeClasse = invoc.getClass().getName();
        long diferenca = System.nanoTime() - millis;
        System.out.println("A classe "+nomeClasse+
            " demorou " + diferenca + " nano segundos");
        return diferenca;
    }
}
```

O método principal dessa classe invoca o método para execução do teste para cada uma das implementações de `InvocadorMetodo`. Adicionalmente, ele faz a divisão do tempo que cada implementação que utilizou reflexão pelo tempo de referência da implementação que invocou o método diretamente. Dessa forma, conseguimos ver quantas vezes mais a invocação com reflexão em cada caso demora. O resultado que foi impresso no console quando executei o teste em minha máquina está apresentado na listagem a seguir.

Listagem 1.14 - Resultado da execução da comparação de desempenho:

```
A classe InvocadorNormal demorou 2334000 nano segundos
A classe InvocadorReflexao demorou 90823000 nano segundos
38.913024850042845 vezes mais que o normal
A classe InvocadorReflexaoCache demorou 6934000 nano segundos
2.9708654670094257 vezes mais que o normal
```

Os resultados em nanosegundos acabam sendo um pouco confusos pois são números muito grandes. Andando seis casas decimais a esquerda temos o resultado em milissegundos, que foi aproximadamente 2 ms, 90 ms e 7 ms para as respectivas classes. Porém, esse resultado pode variar de acordo com a máquina e não é tão interessante quanto a proporção entre esses valores. Pelo que foi impresso no console, podemos observar qua quando a recuperação do método é incluída em cada iteração, o tempo é cerca de 39 vezes maior. Porém, quando o método é recuperado apenas uma vez, essa diferença cai significativamente, sendo aproximadamente apenas 3 vezes maior.

Reflexões sobre a perda de desempenho com reflexão

"Escreva seu código da forma mais simples, mais fácil de manter e mais eficiente possível. Só depois faça a medição de seu desempenho, encontre onde estão os problemas e comece a introduzir otimizações."
– Danny Simmons

Incluí a discussão sobre desempenho logo no primeiro capítulo do livro pois se você vai utilizar reflexão em alguma solução, você precisa saber o preço que vai estar pagando. O fato de uma invocação de método com reflexão demorar cerca de três vezes mais do que uma invocação direta do método, isso não significa que o tempo

de execução de um algoritmo irá triplicar se utilizar reflexão. É preciso avaliar cada caso e levar em consideração o impacto que isso terá no resultado final.

Se observarmos o valor bruto obtido com a invocação de um método por reflexão, podemos ver que o valor de uma única invocação é na ordem de nanosegundos. Isso é um valor bem pequeno se compararmos com o tempo para execução de tarefas de entrada e saída, como leitura de um arquivo, acesso a banco de dados e envio de dados pela rede. Em outras palavras, em relação ao tempo total de execução de uma funcionalidade que envolver funções de entrada e saída, a perda com o uso de reflexão pode ser desprezível.

Essa diferença no desempenho pode se tornar significativa, quando uma operação realizada com reflexão é executada dentro de uma loop repetidas vezes. Nesse caso, a pequena diferença multiplicada por milhares de vezes pode se transformar em um gargalo significativo. Porém, mesmo nesses casos, muitas vezes é possível fazer otimizações. Pelo exemplo apresentado, ficou claro que a recuperação da representação do método leva mais tempo do que sua execução. No caso de uma invocação repetida de métodos por reflexão, uma otimização seria armazenar previamente em uma variável os métodos que seriam invocados.

Tomando como exemplo a geração do mapa de propriedades, imagine que em uma aplicação essa transformação precise ser feita em uma lista com milhares de objetos. Uma possível otimização seria, ao invés de buscar os métodos getters da classe durante a criação do mapa, fazer isso previamente e reutilizar para todos os objetos. Mesmo que na lista houvessem objetos de diferentes classes, essas informações poderiam ser armazenadas por classe e reaproveitadas para todas as suas instâncias.

Um exemplo desse tipo de otimização de desempenho foi feito com o gerador de mapas na próxima listagem. Ao invés de receber o objeto, recuperar sua classe e executar o processamento, essa nova implementação recebe a classe no construtor e já cria um mapa com as propriedades e os métodos para para serem utilizados posteriormente. Dessa forma, ao executar o gerador de mapas ele irá verificar inicialmente se o objeto passado realmente é daquela classe e depois irá utilizar os métodos armazenados no mapa. Isso evitará o overhead da obtenção dos métodos, tornando a geração do mapa mais eficiente se executado para diversos objetos da mesma classe.

Listagem 1.15 - Gerador de mapas que faz cache dos métodos:

```
public class GeradorMapaPerformance {
```

```
    private Map<String, Method> propriedades = new HashMap<>();
    private Class<?> classe;

    public GeradorMapaPerformance(Class<?> classe) {
        this.classe = classe;
        for (Method m : classe.getMethods()) {
            if (isGetter(m)) {
                String propriedade = null;
                if (m.isAnnotationPresent(NomePropriedade.class)) {
                    propriedade = m.getAnnotation(NomePropriedade.class)
                            .value();
                } else {
                    propriedade = deGetterParaPropriedade(m.getName());
                }
                propriedades.put(propriedade, m);
            }
        }
    }
    public Map<String, Object> gerarMapa(Object o) {
        if (!classe.isInstance(o)) {
            throw new RuntimeException("O objeto não é da classe"
                    + classe.getName());
        }
        Map<String, Object> mapa = new HashMap<>();
        for (String propriedade : propriedades.keySet()) {
            try {
                Method m = propriedades.get(propriedade);
                Object valor = m.invoke(o);
                mapa.put(propriedade, valor);
            } catch (Exception e) {
                throw
                    new RuntimeException("Problema ao gerar o mapa", e);
            }
        }
        return mapa;
    }
}
```

Como a frase no início dessa seção sugere, inicialmente deve-se desenvolver o código da forma mais limpa e simples possível. Em seguida, caso os requisitos de desempenho não estejam sendo atendidos, deve-se procurar onde está o gargalo e

realizar a otimização daquela parte do código. A otimização de desempenho precoce é uma má prática, pois pode focar esforços na otimização de uma parte do código que não é significativa no panorama geral da aplicação. Adicionalmente, essa otimização pode ter efeitos colaterais negativos no código, como prejudicar a sua legibilidade ou sua flexibilidade.

Esse livro irá focar no uso de técnicas de reflexão para a elaboração de soluções flexíveis e inteligentes, com potencial de simplificar a criação de aplicações e aumentar a produtividade da equipe. O foco não será na perda ou ganho de desempenho trazido por cada uma das técnicas, a não ser que isso seja um fator relevante no tópico que estiver sendo apresentado. Porém, sugere-se que quando a reflexão for utilizada em uma solução, que os requisitos de desempenho sejam verificados e que técnicas de otimização sejam utilizadas quando necessário.

1.5 Considerações finais

Esse primeiro capítulo teve o objetivo de apresentar o conceito de reflexão computacional e utilização de metadados, mostrando cenários em que as técnicas de orientação a objetos não são suficientes para criar uma solução reutilizável e inteligente. Para os que não se contentam se não verem um pouco de código, foi apresentado um exemplo simples de como a reflexão e o uso de metadados podem auxiliar na criação de uma solução. Para os que olham para reflexão como se fosse um bicho de sete cabeças, peço que não se assustem com o código, pois tudo que foi utilizado será explicado com bastante detalhe nos próximos capítulos. Se desejarem, poderão revisitar esse exemplo mais tarde depois de lerem os dois próximos capítulos.

Por fim, esse capítulo falar sobre uma questão delicada na utilização de reflexão, que é o seu *trade-off* em relação ao desempenho. Como foi mostrado, a flexibilidade do uso dessa técnica tem seu custo em termos de desempenho, e é por isso que é importante ter ciência disso para utilizá-la de forma adequada. Um outro efeito do uso da reflexão é o aumento da complexidade do código, porém isso é algo que pode ser facilmente controlado encapsulando seu uso em classes e componentes que são utilizados pelo resto do código. Afinal, quantos frameworks que utilizavam reflexão você já utilizou sem nem saber como ele estava fazendo aquelas coisas?

Capítulo 2

Java Reflection API

"Diga espelho meu se há na avenida alguém mais feliz que eu"
– "É Hoje", Didi e Maestrinho

Reflexão é considerado um tema avançado, pois trabalha com metaprogramação, ou seja, é um código que trabalha com o próprio código. Por esse motivo, é um tema evitado e temido por muitos programadores. Se por um lado identificar as situações adequadas para o uso dessa técnica realmente não é fácil e exige uma certa experiência, a API Reflection do Java para manipular essas estruturas não é difícil.

Por exemplo, imagine que você esteja começando a trabalhar com uma API para manipular objetos em 3 dimensões. Certamente essa API exigirá uma certa curva de aprendizado até que se obtenha proficiência nela. Porém a dificuldade não é saber quais são as classes e quais métodos chamar, mas sim compreender os conceitos que ela representa para que se saiba como utilizá-la de forma adequada. Se você já trabalhou com objetos em 3 dimensões com outra ferramenta que utilizava os mesmo conceitos, certamente essa curva de aprendizado será muito mais suave. A maior

dificuldade em se aprender uma API está na compreensão dos conceitos que ela representa e não em saber quais são seus métodos e classes.

A API Reflection da linguagem Java trabalha com conceitos familiares a qualquer programador, que são os próprios elementos da linguagem. Uma classe, por exemplo, possui informações como sua superclasse, suas interfaces, seu nome, seu pacote, seus métodos e seus atributos. Você pode fazer com ela, o que se faz com uma classe, instanciar. Da mesma forma, um método possui retorno, nome, parâmetro, modificadores e exceções, e o que se pode fazer com ele é invocar. Por isso, ao começar a trabalhar com a API Reflection, qualquer programador se sente em casa, trabalhando com conceitos que já são familiares.

Esse capítulo tem o objetivo de apresentar as principais funcionalidades da API Reflection. Não se tem a intenção de ser completo e cobrir cada método e cada classe, porém a ideia é apresentar as principais classes com suas respectivas funcionalidades, mostrando através de exemplos como podem ser utilizadas. Ao fim do capítulo, é apresentado um exemplo mais completo que utiliza diversas funcionalidades que foram apresentadas.

2.1 Obtendo informações sobre as classes

"Informação não é conhecimento"
– Albert Einstein

Quando se começa a desenvolver um software em Java, o ponto inicial é sempre uma classe. Para trabalhar com reflexão não é diferente, pois a classe principal que é o ponto de partida para obtermos informações sobre os elementos de um programa é a classe `java.lang.Class` que representa justamente uma classe. Sendo assim, o primeiro passo para começar a trabalhar com reflexão é obtermos a instância de `Class` da classe que queremos trabalhar com ela.

Para entendermos como isso funciona, considere um objeto qualquer de uma aplicação. Esse objeto é descrito pela sua classe, que determina quais são os seus métodos e seus atributos, e por isso podemos dizer que a classe possui as metainformações a respeito dos objetos dela. Porém que tipo de coisa essa classe pode ter? Se a classe descreve o objeto, quem é descreve a classe?

Aí é que entra a instância da classe `Class` que possui todas as informações de uma classe. Essa instância possui as metainformações, não do objeto, mas da classe. Como quais são seus atributos, quais são os seus métodos e qual é sua superclasse e suas interfaces. E daí surge mais uma dúvida: quem descreve a instância de `Class`?

Como vimos, os objetos são descritos por suas classes, então essa instância é descrita pela classe `Class`. E quem descreve a classe `Class`? Uma instância da própria classe `Class`!

Eu sei que isso tudo pode causar um grande nó na nossa cabeça, e a Figura 2.1 mostra de forma esquemática o que foi descrito nos parágrafos anteriores. O que é importante entender aqui é que quando trabalhamos com reflexão, estamos trabalhando um nível acima do que costumamos trabalhar. Enquanto normalmente trabalhamos com objetos que são descritos por classes, com reflexão trabalhamos com classes cujas instâncias descrevem as nossas classes.
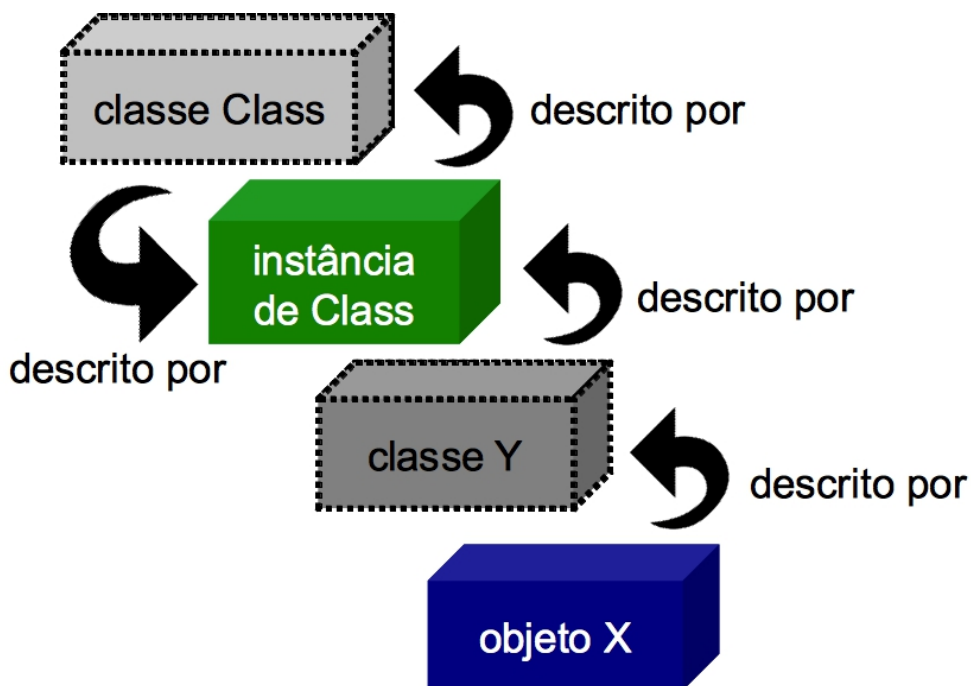
Figura 2.1: Quem possui as metainformações de quem

Utilizando referências estáticas

A forma mais direta de obtermos uma instância de `Class` é através de uma referência estática da classe. Isso é utilizado quando sabemos em tempo de compilação qual classe precisa ser referenciada. Para que isso seja feito, utilizamos o nome da classe seguido por `.class`, o que retorna a instância de `Class` respectiva.

O código apresentado a seguir mostra como as referências estáticas a classes podem ser utilizadas. Inicialmente é criada uma instância de `Class` que recebe a instância referente a classe `String`, da qual é impresso o nome no console. Pode não parecer muito útil utilizar referências estáticas da classe, pois se já sei com qual classe vou trabalhar, para que preciso utilizar reflexão? Porém, essas referências estáticas são muito utilizadas quando precisamos passar uma instância de `Class` como parâmetro para um método, como também é apresentado na listagem.

Listagem 2.1 - Utilizando referências estáticas de Class:

```
public class ReferenciaEstaticaClasse {

    public static void main(String[] args) {
        Class<String> classe = String.class;
        System.out.println(classe.getName());
        imprimeNomeClasse(Integer.class);
    }
    public static void imprimeNomeClasse(Class<?> classe){
        System.out.println("Chamado o método com " +classe.getName());
    }
}
```

O que talvez tenha chamado atenção no exemplo é o fato da classe `Class` possui um tipo genérico. Esse tipo genérico é relativo a classe que está sendo representada. Dessa forma, como será visto posteriormente nesse capítulo, isso permite a inferência de retorno de alguns métodos, o que evita que sejam feitos casts desnecessários.

No caso de não sabermos qual é o tipo, podemos sempre utilizar o wildcard "?" para dizer que não sabemos o tipo de classe que será passado. Um exemplo é o método `imprimeNomeClass()` que recebe como parâmetro o tipo `Class<?>`, significando que qualquer tipo pode ser passado como parâmetro. O uso de tipos genéricos também possibilita que seja feita uma certa restrição no tipo de classe que pode ser passada como parâmetro. Por exemplo, se definirmos o tipo do parâmetro como `Class<? extends Serializable>`, aceitamos como parâmetro somente instâncias de `Class` de tipos que implementam a interface `Serializable`.

Apesar de tipos primitivos em Java não serem considerados classes, é possível obter uma representação de `Class` para eles. Para isso, a referência estática desses tipos pode ser utilizada, como por exemplo `int.class` e `char.class`. Até mesmo o `void` possui uma representação de Class, sendo obtido através da expres-

são `void.class`. Na seção 1.3, onde fizemos a geração do mapa, essa referência foi utilizada no método `isGetter()` para verificar se ele possuía retorno.

Recuperando a classe de um objeto

Uma outra forma muito comum de se obter uma instância de `Class` é através de um objeto dessa classe. A classe `Object` define o método `getClass()` que retorna a representação da classe daquele objeto. A recuperação da `Class` dessa forma é muito comum quando se recebe o objeto como parâmetro e deseja-se informações sobre sua classe. Um exemplo disso foi mostrado no código da geração do mapa, que recebia como parâmetro um `Object` e utilizava o método `getClass()` para obter a sua classe.

Normalmente, essa forma de recuperar a classe é utilizada em métodos mais gerais que recebem um `Object` como parâmetro e utiliza reflexão para conhecer mais sobre a classe e saber quais métodos invocar ou quais atributos acessar. Isso possibilita que objetos de qualquer classe possam ser passados como parâmetro e utilizados pela lógica do método.

A listagem a seguir mostra um exemplo da recuperação da classe a partir do objeto. O tipo genérico retornado pelo método `getClass()` será sempre `Class<? extends Tipo>`, onde "Tipo" representa o tipo da variável na qual o método está sendo invocado. Isso faz sentido, pois uma variável de um determinado tipo pode armazenar objetos de um subtipo. No exemplo, o objeto é do tipo `Integer`, mas está sendo atribuído a uma variável do tipo `Number`. Sendo assim, apesar do tipo retornado pelo `getClass()` ser `Class<? extends Number>`, ao imprimir o nome do objeto `Class` obtido, será impresso no console "java.lang.Integer".

Listagem 2.2 - Recuperando as classes pelos objetos:

```
public class RecuperandoPeloObjeto {

    public static void main(String[] args) {
        Number object = new Integer(100);
        Class<? extends Number> c = object.getClass();
        System.out.println(c.getName());
        //...
    }
}
```

Uma String com o nome da classe

A última forma que será apresentada de se obter uma instância de `Class` é através de um `String` com o nome completo da classe. Essa forma é muito útil para obter nomes de classes de arquivos de configuração e instanciar essas classes a partir da reflexão. Para fazer isso, é preciso chamar o método estático `forName()` da classe `Class` passando o nome da classe como parâmetro. Ao fazer isso, a classe será procurada pela máquina virtual, carregada caso isso ainda não tenha sido feito e retornada. A chamada desse método pode lançar um `ClassNotFoundException` caso a classe não seja encontrada.

USO DO CLASS.FORNAME() NO JDBC

Talvez alguns sintam alguma familiaridade com a chamada `Class.forName()`, pois já viram isso ser utilizado em algum lugar. Um uso comum dessa chamada é para carregar o driver JDBC para o acesso ao banco de dados. Muitos desenvolvedores acabam utilizando essa chamada como parte de uma receita de bolo para acessar um banco de dados, sem saber exatamente o que ela está fazendo. Nesse caso, a chamada faz com que o driver de acesso ao banco seja carregado pela máquina virtual. O carregamento dessa classe resulta na execução de um bloco estático que registra o driver na classe `DriverManager`, o que faz com que as URLs de conexão daquele banco sejam repassadas para aquela classe.

Para exemplificarmos o uso dessa abordagem, a listagem a seguir mostra uma classe que lê um arquivo de propriedades (o qual em cada linha possui propriedade = valor), com interfaces e suas respectivas implementações. Nesse arquivo, as propriedades seriam os nomes das interfaces e o valor o nome de sua respectiva implementação que deve ser utilizada na aplicação. A classe `FornecedorImplementacoes` recebe o nome do arquivo no construtor e utiliza a classe `Properties` para fazer a leitura do arquivo. Para cada propriedade, o `Class.forName()` é utilizado para obter instância de `Class` da interface e sua respectiva implementação, adicionando ambas em um mapa para posterior recuperação.

Listagem 2.3 - Classe que lê arquivo de propriedades com interfaces e implementações:

```
public class FornecedorImplementacoes {

    private Map<Class<?>, Class<?>> implementacoes = new HashMap<>();

    public FornecedorImplementacoes(String nomeArquivo)
                    throws IOException, ClassNotFoundException{
        Properties p = new Properties();
        p.load(new FileInputStream(nomeArquivo));
        for(Object interf : p.keySet()){
            Class<?> interfType = Class.forName(interf.toString());
            Class<?> implType = Class.forName(p.get(interf).toString());
            implementacoes.put(interfType, implType);
        }
    }
    public Class<?> getImplementacao(Class<?> interf){
        return implementacoes.get(interf);
    }
}
```

A classe `FornecedorImplementacoes` pode ser utilizada para configurar em uma aplicação qual a implementação de cada interface. O método `getImplementacao()` recebe como argumento a interface e retorna a classe da implementação. O código a seguir mostra como a classe seria utilizada. Ela é criada passando como parâmetro o arquivo "implementacoes.prop" e tenta recuperar a implementação de uma interface chamada `DAO`. Caso a implementação seja retornada, o nome da classe será impresso no console, e caso contrário a mensagem de erro será exibida.

Listagem 2.4 - Exemplo de uso da classe FornecedorImplementacoes:

```
public static void main(String[] args){
    try {
        FornecedorImplementacoes f =
            new FornecedorImplementacoes("implementacoes.prop");
        Class<?> impl = f.getImplementacao(DAO.class);
        System.out.println("Implementação recuperada: "+impl.getName());
    } catch (ClassNotFoundException | IOException e) {
```

```
        System.out.println(
            "Problemas ao obter implementacões:" + e.getMessage());

    }
}
```

2.2 Trabalhando com classes

Depois de obter a representação de uma classe através de uma instância de `Class`, é possível obter diversas informações além de criar instâncias. Como foi dito na introdução, a API Reflection da linguagem Java não permite que as classes existentes sejam modificadas ou manipuladas, sendo todas as informações somente para leitura. O que pode ser manipulado a partir da API Reflection são os objetos dessas classes, nos quais os atributos podem ser modificados e os métodos podem ser invocados. Nessa seção será visto quais informações podem ser extraídas de uma classe e quais são as formas para instanciá-la através de reflexão.

Acessando informações de classes

Através de uma instância de um objeto do tipo `Class` é possível recuperar diversas informações de uma classe. Como `Class` é utilizada para a representação de qualquer tipo, mesmo que esse não seja exatamente uma classe, existem métodos que podem ser utilizados para verificar o que aquela instância está representando, como `isInterface()`, `isPrimitive()`, `isEnum()` e `isArray()`. A recuperação de métodos e atributos será abordada nas seções seguintes desse capítulo.

A recuperação dos modificadores de uma classe já é um pouquinho mais complicada, pois o método `getModifiers()` retorna um inteiro que possui uma representação de quais são os modificadores da classe. Esses modificadores incluem, para uma classe, `public`, `protected`, `private`, `final`, `static`, `abstract` e `interface`. Para métodos e atributos a recuperação de modificadores é similar, porém os atributos também possuem modificadores como `volatile` e `transient` e os métodos podem possuir os modificadores `synchronized`, `strictfp` e `native`. Esses outros modificadores também são incluídos quando aplicável.

Para decodificar os modificadores, verificando se algum está presente, a classe `Modifier` precisa ser utilizada. Essa classe possui métodos estáticos que recebem o inteiro retornado e verificam se um modificador específico está presente. A listagem a seguir mostra um código que exemplifica esse processo, verificando se a classe é

abstrata.

Listagem 2.5 - Acessando os modificadores da classe:

```
int modificadores = clazz.getModifiers()
if(Modifier.isAbstract(modificadores)){
    //codigo se a classe for abstrata
}
```

Um tipo de verificação que frequentemente precisamos realizar em relação a uma classe é a respeito da sua relação com outros tipos e objetos. A operação `instanceof` é possível ser realizada com um objeto e uma referência estática de uma classe. O método `isInstance()` possui uma lógica similar, porém utilizando uma instância de `Class`. Ele verifica se o objeto passado como parâmetro é uma instância daquela classe.

Outro método relacionado a tipagem, verifica se um objeto da classe passada como parâmetro pode ser convertido na classe onde o método está sendo chamado. O método `isAssignableFrom()` irá retornar `true` caso a classe onde ele está sendo chamado seja uma superclasse, uma interface ou a própria classe que está sendo passada como parâmetro.

Para exemplificar a recuperação de informações da classe, abaixo segue um exemplo de aplicação em que o usuário escreve o nome da classe no console e ele imprime a hierarquia de classes e interfaces. No método `main()` o programa lê o nome da classe utilizando a classe `Scanner` e obtém a instância de `Class` com o método `forName()`. Em seguida, o método `imprimirHierarquia()` é chamado passando a classe como parâmetro.

Listagem 2.6 - Classe que imprime a hierarquia de classes e interfaces:

```
public class InformacaoClasse {

    public static void main(String[] args) {
        System.out.println(
            "Entre com o nome da classe que deseja informação:"
        );
        Scanner in = new Scanner(System.in);
        String nomeClasse = in.nextLine();
        try {
            Class<?> c = Class.forName(nomeClasse);
```

```
            imprimirHierarquia(c, 1);
        } catch (ClassNotFoundException e) {
            System.out.println(
                "A classe "+nomeClasse+" não foi encontrada"
            );
        }
        in.close();
    }
    private static void imprimirHierarquia(Class<?> c, int nivel){
        List<Class<?>> lista = getSuperclasseEInterfaces(c);
        String recuo = "";
        for(int i=0; i<nivel; i++){
            recuo+="   ";
        }
        for(Class<?> clazz : lista){
            System.out.println(recuo+"|-> "+clazz.getName());
            if(clazz != Object.class){
                imprimirHierarquia(clazz, nivel+1);
            }
        }
    }
    private static List<Class<?>> getSuperclasseEInterfaces(Class<?> c){
        List<Class<?>> lista = new ArrayList<>();
        if(c.getSuperclass() != null)
            lista.add(c.getSuperclass());
        lista.addAll(Arrays.asList(c.getInterfaces()));
        return lista;
    }
}
```

O método `imprimirHierarquia()` possui uma lógica recursiva e irá ser chamado novamente para cada classe ou interface encontrados, parando ao se encontrar a classe `Object`. O parâmetro `nivel` representa a profundidade da classe na hierarquia e é utilizado para determinar a quantidade de espaços antes da classe ser impressa no console. Dessa forma, a impressão das classes ficará com um formato similar ao de uma árvore. A cada chamada recursiva, adiciona-se um no parâmetro `nivel`.

O método `getSuperclasseEInterfaces()` retorna uma lista de `Class` com sua superclasse e as interfaces diretamente implementadas por ela. Observe que os métodos `getSuperclass()` e `getInterfaces()` são utilizados para

isso. O método `getInterfaces()` irá retornar as interfaces implementadas caso o `Class` represente uma classe e as interfaces estendidas caso seja uma interface. O código abaixo apresenta a saída do software caso seja fornecida a entrada "java.util.ArrayList".

Listagem 2.7 - Saída do programa para a classe ArrayList:

```
Entre com o nome da classe que deseja informação:
java.util.ArrayList
   |-> java.util.AbstractList
      |-> java.util.AbstractCollection
         |-> java.lang.Object
         |-> java.util.Collection
            |-> java.lang.Iterable
      |-> java.util.List
         |-> java.util.Collection
            |-> java.lang.Iterable
   |-> java.util.List
      |-> java.util.Collection
         |-> java.lang.Iterable
   |-> java.util.RandomAccess
   |-> java.lang.Cloneable
   |-> java.io.Serializable
```

Criando instâncias

Uma classe não é nada sem as suas instâncias! Se queremos carregar classes de arquivos de configuração para utilizarmos na aplicação, é preciso criar instâncias delas. Isso pode ser feito facilmente através do método `newInstance()` de `Class`! Esse método cria uma nova instância da classe a partir de um construtor vazio. Ao invocar esse método existem duas exceções que precisam ser tratadas: `InstantiationException` e `IllegalAccessException`. A primeira será lançada caso não exista um construtor vazio ou a instância de `Class` represente uma classe abstrata, interface, tipo primitivo ou array. A segunda exceção já será lançada caso o construtor não esteja acessível, por exemplo, por ser privado.

Frameworks que pedem construtores sem parâmetros

Existem vários frameworks que em sua documentação dizem que para suas classes poderem ser manipuladas por ele, que precisam possuir um construtor sem parâmetro. Agora você já sabe que isso é porque ele utiliza esse método `newInstance()` para instanciá-la. Repare que isso é comum quando o framework precisa instanciar classes da sua aplicação. Isso é comum, por exemplo, em frameworks web para classes que representam controllers, que são criadas e gerenciadas pelo framework para tratar requisições. Outro exemplo ocorre em frameworks de mapeamento objeto-relacional, os quais precisam instanciar as classes mapeadas para o banco quando informações são recuperadas.

O método `newInstance()` tem um sério problema em relação a exceções! Caso o construtor lance uma exceção, mesmo que checada, essa exceção será propagada diretamente para o método que tentou criar a instância. Isso sem exigir que essa exceção seja tratada! Veja um exemplo disso no código apresentado na listagem a seguir. Quando ele for executado, a `IOException` lançada no construtor será propagada para o método `main()` sem cair em nenhum dos blocos `catch`. Para conseguir tratar essa questão, deve-se utilizar diretamente a classe `Constructor`.

Listagem 2.8 - Tratamento de exceções com o método newInstance():

```
public class CriacaoClasse {

    public CriacaoClasse() throws IOException{
        throw new IOException();
    }
    public static void main(String[] args){
        try {
            CriacaoClasse obj = CriacaoClasse.class.newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Os construtores podem ser obtidos de uma classe através dos métodos `getConstructor()` ou `getConstructors()`. O primeiro método recebe como parâmetro os tipos dos parâmetros do construtor, permitindo recuperar um construtor específico. Esse método utiliza o recurso da linguagem de **varargs** para permitir que se passe diversos parâmetros com os tipos. O segundo método recupera uma lista com todos os construtores da classe. Para utilizar o construtor, basta chamar o método `newInstance()` passando os parâmetros adequados. Da mesma forma que a recuperação do construtor, esse método também utiliza **varargs** para que se possa passar quando parâmetros forem necessários.

Em relação as exceções, a recuperação de um construtor específico pode lançar o erro `NoSuchMethodException` caso aquele construtor não exista. A invocação de um construtor, além das duas exceções que podem ser lançadas pelo `newInstance()` de `Class`, duas outras exceções podem ser lançadas. A exceção `IllegalArgumentException` é lançada quando a quantidade e os tipos dos parâmetros não corresponderem aos do construtor e `InvocationTargetException` é a exceção que encapsula uma exceção que é lançada pelo construtor. Nesse caso, a exceção original pode ser recuperada através do método `getTargetException()`.

O código abaixo mostra o exemplo do uso de um construtor. A classe `UsoConstrutor` possui um construtor que recebe uma `String` como argumento. O método `getConstructor()` é utilizado para recuperar o construtor e o método `newInstance()` para a criação do objeto. Observe que no tratamento de `InvocationTargetException` a exceção original é recuperada e impressa no console.

Listagem 2.9 - Exemplo de criação de instância com construtores:

```
public class UsoConstrutor {

    public UsoConstrutor(String s){
        System.out.println("Construtor invocado com: "+s);
    }
    public static void main(String[] args)
        throws NoSuchMethodException, SecurityException {
        Class<UsoConstrutor> c = UsoConstrutor.class;
        Constructor<UsoConstrutor> constr =
            c.getConstructor(String.class);
        try {
```

```
            UsoConstrutor obj = constr.newInstance("teste");
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            System.out.println(
            "Exceção lançada pelo construtor: "+e.getTargetException());
        }
    }
}
```

Quando criar objetos utilizando reflexão?

A criação de objetos utilizado reflexão é indicada quando não se conhece em tempo de compilação a classe que vai ser instanciada. No caso de haverem diversas implementações para uma classe, mas todas serem conhecidas, um padrão de criação, como Factory Method ou Builder, pode ser utilizado para determinar a classe que será criada. A criação de objetos através de reflexão é indicada quando se deseja que novas classes possam ser criadas e configuradas como plugins de um software ou framework existente.

Melhorando o fornecedor de implementações

A partir do que foi visto nesse seção podemos melhorar a implementação da classe `FornecedorImplementacoes`. A primeira verificação será na leitura do arquivo, onde podemos verificar se a classe utilizada como chave é realmente uma interface ou uma classe abstrata e a classe passada como valor é realmente uma implementação concreta dela. Outra modificação é que ao invés de retornar somente a classe da implementação, vamos ter também métodos para já retornar instâncias das implementações.

Começando pelas verificações, inicialmente vamos criar métodos auxiliares que verificam propriedades das classes, conforme mostrado na listagem a seguir. O método `isInterfaceOuAbstract()` verifica se o `Class` passado como

parâmetro é uma interface ou uma classe abstrata. Observe que a classe auxiliar `Modifiers` foi utilizada para verificar se o modificador `abstract` está presente. Em seguida temos o método `isAbstracaoEImplementacao()` que verifica se a primeira classe é uma abstração, se a segunda classe é uma implementação e se a implementação é do tipo da abstração.

Listagem 2.10 - Métodos auxiliares para verificação:

```
private boolean isInterfaceOuAbstract(Class<?> c){
    return c.isInterface() || Modifier.isAbstract(c.getModifiers());
}
private boolean isAbstracaoEImplementacao
        (Class<?> interf, Class<?> impl){

    return isInterfaceOuAbstract(interf) && !isInterfaceOuAbstract(impl)
            && interf.isAssignableFrom(impl);

}
```

A listagem a seguir mostra como o construtor da classe `FornecedorImplementacoes` pode ser modificado para incluir a verificação. Observe que depois que os objetos `Class` são criados, caso o método `isAsbtracaoEImplementacao()` retorne `false`, uma exceção será lançada para indicar um erro de configuração no arquivo.

Listagem 2.11 - Construtor modificado para fazer as verificações:

```
public FornecedorImplementacoes(String nomeArquivo) throws Exception{
    Properties p = new Properties();
    p.load(new FileInputStream(nomeArquivo));
    for(Object interf : p.keySet()){
        Class<?> interfType = Class.forName(interf.toString());
        Class<?> implType = Class.forName(p.get(interf).toString());
        if(!isAbstracaoEImplementacao(interfType, implType)){
            throw new Exception("Erro na configuração do arquivo " +
                nomeArquivo + " : " + interfType.getName() +
                " não é abstração de " + implType.getName());
        }
        implementacoes.put(interfType, implType);
    }
}
```

A segunda parte que será adicionada no fornecedor de implementações é a que cria os objetos a partir da classe da respectiva implementação. Porém não queremos que seja utilizado apenas o construtor sem parâmetros. O objetivo é que a partir dos parâmetros passados para a criação do objeto, seja encontrado um construtor que possa receber aqueles parâmetros. A listagem a seguir apresenta um método auxiliar que será utilizado para achar um construtor adequado aos parâmetros passados.

Listagem 2.12 - Métodos auxiliar que busca um construtor adequado:

```
private Constructor<?> acharConstrutor(Class<?> c, Object... objs)
        throws Exception{
    for(Constructor<?> constr : c.getConstructors()){
        Class<?>[] params = constr.getParameterTypes();
        if(params.length == objs.length){
            boolean erro = false;
            for(int i=0; i<objs.length && !erro; i++){
                if(!params[i].isInstance(objs[i]))
                    erro = true;
            }
            if(!erro)
                return constr;
        }
    }
    throw new Exception("Construtor não encontrado");
}
```

Inicialmente, o método recupera a lista de construtores a partir do método `getConstructors()`. E seguida, para cada construtor são feitas verificações para ver se ele é adequado aos parâmetros. A primeira questão verificada é se o número de parâmetros do construtor é igual ao número de objetos passados como parâmetro. Em seguida, para cada parâmetro é verificado se o objeto possui o tipo exigido para o parâmetro, utilizando o método `isInstance()`. Caso a verificação seja positiva para todos os parâmetros, o construtor é retornado para o método. Em caso negativo, uma exceção é lançada indicando que um construtor para aqueles parâmetros não foi encontrado. Pela lógica, é possível perceber que o primeiro construtor que se encaixe será retornado.

Para completar essa funcionalidade, a próxima listagem apresenta o método `criarInstancia()` que efetivamente cria o objeto. Esse método inicialmente utiliza o método `getImplementacao()` para recuperar a implementação da

interface ou classe abstrata passada como parâmetro. Em seguida, o método criado na listagem anterior é utilizado para recuperar o construtor, o qual é utilizado para a criação da instância através do método `newInstance()`.

Listagem 2.13 - Criação da instância através do construtor:

```
public Object criarInstancia(Class<?> interf, Object... objs)
        throws Exception{
    Class<?> impl = getImplementacao(interf);
    Constructor<?> constr = acharConstrutor(impl, objs);
    return constr.newInstance(objs);
}
```

Uma ressalva em relação a essa implementação é que ela não irá funcionar caso o construtor receba tipos primitivos como parâmetros. O problema é que o tipo primitivo será convertido para sua respectiva classe **wrapper** ao ser passado para o **varargs** no array de objetos. Dessa forma, a invocação do método `isInstance()` irá retornar `false` por causa dessa conversão. Deixo como exercício aos leitores fazer essa implementação funcionar para construtores que recebem tipos primitivos.

Cuidado com os tipos primitivos!

Os tipos primitivos são uma grande pedra no sapado de quem trabalha com reflexão. Apesar do seu tipo ser representado pela classe `Class`, muitas vezes é preciso fazer a distinção dos tipos primitivos e tratá-los como casos especiais. O exemplo da busca de construtores é um caso em que os tipos primitivos precisariam ser tratados separadamente, trocando seu tipo pelo de uma classe **wrapper** ou utilizando um condicional para cada tipo primitivo. Sendo assim, não se incomode ao ter que criar uma séria de condicionais para tratar de forma especial cada um dos tipos primitivos, pois nos frameworks que desenvolvi precisei fazer isso algumas vezes.

2.3 Manipulando objetos

Nas seções anteriores vimos como recuperar a referência de uma classe, como acessar suas informações e como criar novas instâncias. Essa seção mostra como é possível

manipular os objetos com reflexão através do acesso aos seus atributos e da invocação de seus métodos. Existem algumas questões mostradas para uma classe, cuja recuperação é similar quando lidamos com atributos e métodos. Por exemplo, a recuperação de modificadores em métodos e atributos também é feita com o método `getModifiers()` e também é decodificada com a classe `Modifier`. Essas questões que são muito similares não serão apresentadas novamente.

Acessando e alterando valores de atributos

Através da API Reflection é possível recuperar os atributos de uma classe, que são representados pela classe `Field`. A partir dessas referências dos atributos seus valores podem ser acessados e modificados. Existem dois métodos para retornar a lista de atributos de um classe: `getFields()` que retorna todos os atributos públicos e `getDeclaredFields()` que retorna todos os atributos que foram declarados naquela classe. Para exemplificarmos a diferença entre esses dois métodos, considere o código apresentado na listagem seguinte.

Listagem 2.14 - Programa que mostra a diferença entre getFields() e getDeclaredFields():

```
public class SuperclasseAtributo {
    private int atributoSuperclasseUm;
    public String atributoSuperclasseDois;
}

public class AcessoAtributo extends SuperclasseAtributo{

    private int atributoUm;
    public String atributoDois;

    public static void main(String[] args) {
        System.out.println("Retornado pelo getFields()");
        for(Field f : AcessoAtributo.class.getFields()){
            System.out.println(
                "- "+f.getType().getSimpleName() + " " + f.getName());
        }
        System.out.println("Retornado pelo getDeclaredFields()");
        for(Field f : AcessoAtributo.class.getDeclaredFields()){
            System.out.println(
                "- "+f.getType().getSimpleName() + " " + f.getName());
```

```
        }
    }
}
```

O resultado da execução desse programa está apresentado na listagem a seguir. Observe que o método `getFields()` retorna todos os atributos públicos, incluindo tanto os da própria classe quanto os da superclasse. Já o método `getDeclaredFields()` retorna todos os atributos declarados na própria classe, incluindo os que são privados, porém não incluindo nenhum declarado na superclasse. Se quisermos todos os atributos, incluindo os privados da classe e das superclasses, precisamos ir percorrendo a hierarquia de classes e recuperando os atributos de cada uma com `getDeclaredFields()`. De forma similar, se quisermos acessar um atributo específico da classe, também temos dois métodos com comportamento análogo, sendo eles `getField()` e `getDeclaredField()`.

Listagem 2.15 - Saída do programa com a diferença de getFields() para getDeclaredFields():

```
Retornado pelo getFields()
- String atributoDois
- String atributoSuperclasseDois
Retornado pelo getDeclaredFields()
- int atributoUm
- String atributoDois
```

As duas ações que podemos fazer com o atributo de um objeto são a recuperação e a modificação de seu valor. A classe `Field` possui dois métodos que executam respectivamente essas ações, sendo eles o método `get()` e o método `set()`. Ambos os métodos recebem como parâmetro o objeto no qual esse atributo deve ser acessado. Para exemplificar o acesso dos valores de um atributo, considere a classe chamada `ExemploClasse` representada na próxima listagem. Observe que ela possui um atributo público, um atributo privado e um atributo estático e público.

Listagem 2.16 - Exemplo de classe com atributos a serem acessados por reflexão:

```
public class ExemploClasse {
    public String publico;
    private String privado;
```

```
    public static String estatico;
}
```

A listagem a seguir, mostra um código que acessa cada um desses atributos, inserindo como valor seu próprio nome e em seguida recuperando esse valor e imprimindo-o no console. O método `escreverLerAtributo()` recebe como parâmetro o `Field` e o objeto no qual ele precisa ser recuperado. Esse método usa os métodos `set` e `get` para primeiro atribuir o nome do atributo como valor e depois recuperá-lo. A exceção `IllegalArgumentException` irá acontecer quando um objeto de um tipo errado for passado para os métodos de `Field`, o que não precisamos nos preocupar muito nesse caso. Por outro lado, a exceção `IllegalAccessException` irá ocorrer caso ocorra a tentativa de acesso a um atributo cujo acesso não é permitido.

Listagem 2.17 - Acesso a diferentes tipos de atributos por reflexão:

```
public class EscrevendoLendoAtributos {

    public static void main(String[] args)
            throws NoSuchFieldException, SecurityException {
        ExemploClasse instancia = new ExemploClasse();
        Class<?> clazz = instancia.getClass();
        escreverLerAtributo(clazz.getField("publico"), instancia);
        escreverLerAtributo(
                clazz.getDeclaredField("privado"), instancia);
        escreverLerAtributo(clazz.getField("estatico"), null);
    }
    public static void escreverLerAtributo(Field f, Object instancia){
        try {
            f.set(instancia, f.getName());
            Object valor = f.get(instancia);
            System.out.println("Escrito e lido o atributo = "+valor);
        } catch (IllegalArgumentException e) {
            System.out.println("Problemas ao acessar atributo "
                +f.getName()+": "+e.getMessage());
        } catch (IllegalAccessException e) {
            System.out.println("Problemas de acesso no atributo "
                +f.getName()+": "+e.getMessage());
        }
    }
```

```
}
```

O método `main()` cria uma instância da classe `ExemploClasse` e recupera sua respectiva instância de `Class`, a partir da qual os atributos são recuperados. Observe que o atributo privado precisa ser recuperado com `getDeclaredField()`, pois o método `getField()` só irá encontrar os atributos públicos. Outra observação importante é que não é necessária uma instância para que um atributo estático seja acessado, visto que ele está vinculado a classe.

Ao executar esse programa, o atributo privado não conseguirá ser acessado e a exceção `IllegalAccessException` será lançada. Caso esse código estivesse na própria classe `ExemploClasse`, esse atributo estaria acessível e o erro não aconteceria. É importante ressaltar que por mais que os atributos privados estejam acessíveis para ser recuperados via reflexão, nem sempre eles poderão ser acessados. Isso vai depender de onde o código está sendo executado e as mesmas restrições que se aplicariam ao acesso direto, também se aplicam a reflexão. O último capítulo desse livro mostra como essa regra pode ser burlada para permitir o acesso de membro privados através da API Reflection.

Invocando métodos

Ao falar sobre métodos, não sobrou muita coisa a ser dita, pois a recuperação dos métodos é similar a recuperação de atributos e a invocação dos métodos também não é muito diferente da chamada de construtores. A recuperação de um conjunto de métodos é similar a recuperação de atributos, havendo dois métodos chamados `getMethods()` e `getDeclaredMethods()`. Em Java, um método é identificado unicamente pelo seu nome e pelos seus parâmetros, pois podem existir métodos com o mesmo nome e que recebe parâmetros diferentes. Dessa forma, os métodos `getMethod()` e `getDeclaredMethod()` recebem como primeiro parâmetro o nome e em seguida um número variável de objetos do tipo `Class` para representar os tipos dos parâmetros.

Os métodos são representados pela classe `Method` e possuem diversas informações interessantes de serem recuperadas, que podem ser utilizadas quando procuramos um método adequado para ser invocada. O método `getReturnType()` retorna o tipo do retorno do método, que pode ser utilizado para saber o que deve se esperar de sua invocação. Se o método em questão não possuir retorno, esse método irá retornar o tipo `void.class`. Outro método interessante é o `getParameterTypes()` que retorna um array com os tipos dos parâme-

tros. No exemplo apresentado na introdução desse livro, esses dois métodos foram utilizados para verificar se o método começado com "get" retornava algo diferente de `void` e não possuía parâmetros. Uma outra informação importante dos métodos são os tipos de anotações lançados, o que pode ser recuperado com `getExceptionTypes()`.

A invocação de métodos é feita a partir do método `invoke()`, que recebe como primeiro parâmetro o objeto em que o método deve ser invocado e em seguida os valores que devem ser passados como argumento. Esse método irá retornar o retorno do método que está sendo invocado. Assim como na invocação de construtores, caso uma exceção seja lançada pelo método, será lançada uma exceção do tipo `InvocationTargetException`, através da qual a exceção original pode ser recuperada com o método `getTargetException()`. E assim como o acesso a atributos estáticos, um método estático pode ser invocado passando como `null` o primeiro parâmetro, o qual representa o objeto onde o método deve ser invocado.

Para exemplificar a invocação de métodos por reflexão, vamos mostrar um exemplo onde o usuário escolhe a classe e o método que ele deseja executar. A partir dessa escolha, o programa irá verificar quais são os parâmetros necessários para para execução daquele método e pedir que o usuário entre com cada um deles. Por fim, o método será invocado com os parâmetros passados e o retorno será exibido no console. É importante ressaltar aqui que o objetivo desse exemplo é simplesmente ilustrar com código a invocação de métodos via reflexão, e não realmente suportar a invocação de qualquer tipo de método via linha de comando.

Para começar o exemplo, a partir da entrada do usuário do nome do método precisaremos procurar esse método entre os métodos da classe. Como não é possível recuperar um método diretamente de `Class` apenas pelo seu nome, pois os tipos dos parâmetros seriam também necessários, será necessário percorrer toda lista de métodos procurando um método entrado pelo usuário. Sendo assim, o método auxiliar apresentado na listagem a seguir realiza essa procura de método em uma classe, retornando o primeiro que for encontrado com o nome passado como parâmetro.

Listagem 2.18 - Método auxiliar que procura o método pelo seu nome:

```
private static Method acharMetodoPeloNome(Class<?> c, String nome)
        throws Exception{
    for(Method m : c.getMethods()){
        if(m.getName().equals(nome)){
```

```
            return m;
        }
    }
    throw new Exception("Método "+nome+" não encontrado");
}
```

Na listagem a seguir, está apresentado o corpo principal do programa que executa o método de acordo com as entradas do usuário. Inicialmente ele solicita ao usuário o nome da classe e em seguida o nome do método. O método será procurado utilizando o método `acharMetodoPeloNome()` apresentado na listagem anterior.

A partir do método recuperado, os tipos e a quantidade dos parâmetros são recuperados para serem solicitados ao usuário. Para cada parâmetro, é solicitado para o usuário que entre com um valor. Esse valor é lido como uma `String` e é utilizado para criação do objeto que será o valor do parâmetro, através da invocação de um construtor que recebe uma `String` como argumento. Sendo assim, se o tipo do parâmetro for `Integer`, será invocado seu construtor passando a `String` lida como parâmetro.

Depois que todos os parâmetros forem lidos, o método é invocado utilizando método `invoke()` e seu resultado é impresso no console.

Listagem 2.19 - Código que executada um método de acordo com as entradas do usuário:

```
public class ExecutaMetodo {

    public static void main(String[] args) throws Exception{
        System.out.println(
            "Entre com o nome da classe "
            + "com método que deseja executar:");
        Scanner in = new Scanner(System.in);
        String nomeClasse = in.nextLine();
        Class<?> c = Class.forName(nomeClasse);
        System.out.println("Entre com o nome do método:");
        String nomeMetodo = in.nextLine();
        Method m = acharMetodoPeloNome(c, nomeMetodo);
        Object[] params = new Object[m.getParameterTypes().length];
        for(int i=0; i<params.length; i++){
            Class<?> paramType = m.getParameterTypes()[i];
```

```
                System.out.println("Parametro "+(i+1)+
                                   " ("+paramType.getName()+")");
                String valor = in.nextLine();
                params[i] = paramType.
                            getConstructor(String.class).newInstance(valor);
            }
            Object retorno = m.invoke(c.newInstance(), params);
            System.out.println("O método retornou: " + retorno);
            in.close();
    }
}
```

Vale ressaltar que diversas coisas poderiam dar errado na execução desse método, que está simplesmente declarando que pode jogar uma exceção e não estão sendo tratadas da forma apropriada. Ele assume, por exemplo, que a classe entrada pelo usuário possuirá um construtor vazio e que os tipos dos parâmetros terão construtores que recebem uma `String` como parâmetro. Em uma aplicação real, seria importante tratar os casos em que essas premissas não são verdade. Por exemplo, se quiséssemos dar suporte a parâmetros que são tipos primitivos, precisaríamos de um tratamento especial para cada um deles.

A classe apresentada a seguir, chamada `Utilitaria`, possui um método que será utilizado para testarmos a execução de métodos através do programa desenvolvido. O método `repetir()` dessa classe recebe três parâmetros e possui uma lógica bem simples. Ele irá retornar uma `String` em que o parâmetro `base` é repetido segundo o parâmetro `vezes`, separado pelo parâmetro `divisor`.

Listagem 2.20 - Classe utilizada para testar a execução de métodos:

```
public class Utilitaria {

    public String repetir(String base, String divisor, Integer vezes){
        String retorno = base;
        for(int i=1; i<vezes; i++){
            retorno += divisor+base;
        }
        return retorno;
    }
}
```

A listagem a seguir apresenta a saída do console a partir da execução dessa

programa. Observe que o nome da classe `Utilitaria` e o nome do método `repetir()` são entrados como as duas entradas iniciais do usuário. Em seguida, o programa pede o valor para cada parâmetro, mostrando o tipo esperado para ele. Por fim, depois de entrar com valores para os três parâmetros, é exibido o retorno do método como esperado.

Listagem 2.21 - Saída do programa que executa o método repetir():

```
Entre com o nome da classe com método que deseja executar:
org.casadocodigo.Utilitaria
Entre com o nome do método:
repetir
Parametro 1 (java.lang.String)
teste
Parametro 2 (java.lang.String)
##
Parametro 3 (java.lang.Integer)
4
O método retornou: teste##teste##teste##teste
```

Quando executar métodos por reflexão?

A orientação a objetos provê meios de você invocar métodos em um objeto que você não conhece previamente. Isso pode ser feito definindo uma abstração, como uma interface ou uma classe, que possua essa método e utilizando o polimorfismo para invocar esse método em qualquer objeto que obedeça a essa abstração. Nesses casos, quando é possível ter uma abstração que representa o método que deve ser invocado, a reflexão não precisa ser utilizada. Por outro lado, se você precisa lidar com classes que possuem métodos diferentes e não faz sentido compartilharem uma mesma abstração, essa é a deixa para a invocação desses métodos por reflexão. Um bom exemplo desse caso são os Java Beans, que possuem diversos métodos getter e setter diferentes e não é possível criar uma interface comum que capture essa característica.

2.4 Procurando métodos e atributos para validação

Durante as seções anteriores foram apresentados vários exemplos que demonstraram o funcionamento da API Reflection. O objetivo dessa seção é consolidar esse conhecimento através de um exemplo mais realista. O requisito principal é criar uma classe que invoca para um determinado objeto rotinas de validação. Essas rotinas de validação podem ser métodos definidos na própria classe ou através de atributos que implementam a interface `Validador`, que recebe o próprio objeto como parâmetro e está apresentada na listagem a seguir.

A ideia é que os validadores armazenados em atributos possam ser reutilizados em diferentes classes e que os definidos em métodos possuam validações mais específicas da própria classe. O objetivo é que na validação todos os validadores presentes na classe sejam invocados e, caso existam erros, uma lista com eles seja retornada.

Listagem 2.22 - Interface para atributos com validadores:

```
public interface Validador {
    public void validar(Object o) throws Exception;
}
```

Vamos começar o exemplo pelo método que executa os validadores que estão presentes nos atributos. Esse método deve percorrer os atributos da classe e buscar pelos que implementam a interface `Validador`. A medida que esses validadores forem sendo encontrados, eles devem ir sendo executados e, caso uma exceção seja lançada, esse erro deve ser armazenado e retornado. A implementação desse método está apresentada na listagem a seguir. Um fator a ser observado é que o método `validar()` não é invocado utilizando reflexão e sim polimorfismo através da própria interface. Como o tipo do atributo era conhecido, não foi necessário utilizar reflexão para essa invocação.

Listagem 2.23 - Método que executa os validadores em atributos da classe:

```
private List<Exception> chamarValidadores(Object obj, Class<?> clazz){
    List<Exception> erros = new ArrayList<>();
    for(Field f : clazz.getFields()){
        if(Validador.class.isAssignableFrom(f.getType())){
```

```
            try {
                Validador v = (Validador) f.get(obj);
                v.validar(obj);
            } catch
                (IllegalArgumentException | IllegalAccessException e) {
                throw new RuntimeException(e);
            } catch (Exception e) {
                erros.add(e);
            }
        }
    }
    return erros;
}
```

Em seguida, foi criado o método que invoca os métodos de validação presentes na própria classe. Por convenção, esses métodos devem começar com a palavra "validar" e não possuírem parâmetros, visto que como estão na própria classe podem acessar seus atributos. A listagem a seguir mostra o método `chamarMetodosValidacao()`, que percorre os métodos da classe buscando os métodos de validação e invocando-os. Observe que caso o método `invoke()` lance uma `InvocationTargetException`, essa é capturada pela cláusula `catch` e a exceção original é recuperada pelo método `getTargetException()` e adicionada na lista de erros.

Listagem 2.24 - Método que chama os métodos validadores da classe:

```
private List<Exception> chamarMetodosValidacao
                            (Object obj, Class<?> clazz){
    List<Exception> erros = new ArrayList<>();
    for(Method m : clazz.getMethods()){
        if(m.getName().startsWith("validar")
                && m.getParameterTypes().length == 0){
            try {
                m.invoke(obj);
            } catch
                (IllegalAccessException | IllegalArgumentException e) {

                throw new RuntimeException(e);

            } catch (InvocationTargetException e) {
```

```
                erros.add((Exception)e.getTargetException());
            }
        }
    }
    return erros;
}
```

Para finalizar precisamos ter uma método principal que invoca os dois métodos e reúne os erros. Para que seja possível fazer isso, precisamos definir uma exceção de validação que reúna esses erros e possa ser lançada a quem invocar o método. Dessa forma, a listagem a seguir mostra a classe `ValidacaoException`, que possui um atributo que recebe uma lista de exceções em seu construtor e permite sua posterior recuperação.

Listagem 2.25 - Exceção que reúne diversos erros de validação:

```
public class ValidacaoException extends Exception {

    private List<Exception> erros;

    public ValidacaoException(List<Exception> erros) {
        this.erros = erros;
    }
    public List<Exception> getErros() {
        return erros;
    }
}
```

Finalmente, a listagem a seguir apresenta o método `validarObjeto()` que orquestra a invocação dos outros dois métodos desenvolvidos. Inicialmente, o método extrai a instância de `Class` do objeto recebido como argumento e cria uma lista para armazenar os erros retornados pelos métodos. Em seguida os métodos são invocados, tendo os erros retornados adicionados na lista. No fim, caso a lista possua algum erro, é criada uma nova `ValidacaoException` com essa lista e lançada para o método que o invocou. Sendo assim, caso o método capture essa exceção, ele terá acessoa a todos os erros lançados pelos validadores.

Listagem 2.26 - Método principal que faz a validação do objeto:

```
public class ValidadorObjetos {
```

```
    public void validarObjeto(Object obj) throws ValidacaoException {
        Class<?> clazz = obj.getClass();
        List<Exception> erros = new ArrayList<>();
        erros.addAll(chamarMetodosValidacao(obj, clazz));
        erros.addAll(chamarValidadores(obj, clazz));
        if(erros.size() > 0){
            throw new ValidacaoException(erros);
        }
    }
}
```

Diferentemente dos exemplos anteriores, onde mostro o método desenvolvido sendo utilizado, dessa vez deixarei isso como exercício para os leitores. Procure criar uma classe que possua tanto métodos de validação quanto atributos que tenha o tipo `Validator`. Tente também criar contraexemplos, ou seja, métodos e atributos que não deveriam ser incluídos e veja se realmente são deixados de fora.

2.5 Coisas que podem dar errado

"Com grandes poderes vêm grandes responsabilidades"
– Ben Parker, tio do Homem-Aranha

Quando trabalhamos com reflexão, existem diversos erros que podem acontecer que normalmente são validados em tempo de compilação. Por exemplo, um método pode ser invocado com quantidade ou tipos de parâmetros inválidos, pode-se tentar acessar um atributo em um objeto de uma classe que não o possui ou mesmo pode-se tentar acessar membros de classe que não estão acessíveis. Durante os exemplos, podemos ver diversas situações em que erros desse tipo eram possíveis e foram tratados.

Ao utilizar reflexão conseguimos uma maior flexibilidade, porém esse poder deve ser utilizado com responsabilidade. Todos esses erros devem ser verificados, tratados e evitados quando possível. Por exemplo, se o código assume que o método ou construtor chamado possui parâmetros de um determinado tipo, procure verificar se essa premissa é verdadeira antes de invocá-lo. Em outras palavras, procure verificar as condições com antecedência e tratar as condições excepcionais, ao invés de simplesmente chamar os métodos e deixar os erros acontecerem.

Quando lidamos com reflexão, a própria estrutura da classe é um parâmetro, então devemos estar preparados para lidar com qualquer tipo de classe. Tipos como

arrays e primitivos costumam exigir tratamento especial em alguns casos. Dessa forma, o código deve considerar e tratar esses casos, e mesmo que não se deseje dar suporte a esses casos, deve-se verificar se a estrutura da classe ou método é adequada e lançar um erro caso não seja. Quando desenvolvi minhas primeiras classes que utilizavam reflexão, me lembro de receber alguns bugs relacionados a estruturas de classe que eu não havia pensado. Sendo assim, criar testes que contemplam diversos tipos de classe é essencial para que se tenha uma boa segurança no código desenvolvido.

Por fim, procure evitar esconder erros do usuário, de forma que sua classe "falhe gentilmente" sem que ele perceba. Imagine que um arquivo de configuração possua uma lista de atributos de uma classe que precisam ser utilizados em um algoritmo que utilize reflexão, um exemplo de "falhar gentilmente" seria simplesmente pular um atributo que não existir na classe, realizando o processamento para os outros. Por mais que isso pode parecer ajudar evitando que o desenvolvedor enfrente uma série de exceções antes de tudo funcionar pela primeira vez, na verdade isso mascara um erro que pode ser difícil de ser detectado depois. Sendo assim, se encontrar alguma coisa que não está correta, lance o erro com a maior quantidade de detalhes possível, de forma a permitir que o desenvolvedor que utilizar a sua classe identifique o que está errado e realize a correção o mais cedo possível.

O capítulo 5 apresenta formas de realizar os testes de classes que utilizam reflexão. Para aqueles que pretendem utilizar reflexão em projetos reais, as técnicas apresentadas nesse capítulo são essenciais para ajudar na validação e verificação desse tipo de código.

2.6 Considerações finais

Esse capítulo apresentou o principal da API Reflection da linguagem Java. Apesar de uma noção inicial já ter sido dada na introdução, esse capítulo foi mais fundo nos detalhes e apresentou os principais detalhes dessa API. O capítulo começou apresentando as formas em que uma referência de uma classe pode ser obtida, sendo essas através de um referência estática, um objeto e uma `String` com seu nome. Através da instância de `Class` foi mostrado como obter informações a respeito da classe e como criar instâncias dela. A partir das instâncias foi mostrado como a reflexão pode ser utilizada para o acesso aos seus atributos e para a invocação de seus métodos.

Em todo capítulo, a cada novo conceito apresentado, víamos um exemplo que demonstrasse de forma simples como ele funcionava. A medida que o capítulo foi

seguindo, os novos exemplos procuravam incluir questões relacionadas a funcionalidades da API apresentadas previamente, mostrando como elas se encaixavam e podiam ser utilizadas em conjunto. No final do capítulo, um exemplo mais próximo do real foi desenvolvido utilizando o que havia sido apresentado previamente.

Para fechar o capítulo, foi ressaltada a importância em se tomar cuidado com as situações excepcionais que podem acontecer ao se utilizar reflexão. Como diversos erros que normalmente são pegos em tempo de compilação podem ocorrer, é importante tomar cuidado para que todas as situações sejam tratadas adequadamente.

Capítulo 3

Metadados e Anotações

"Ao procurar uma citação sobre metadados, só encontrei metadados sobre citações"
– metalinguagem sobre a escrita do livro

O prefixo *meta* é frequentemente utilizado junto com uma palavra para significar que ela está se referindo a si própria. Me lembro das aulas de literatura, onde a professora frequentemente falava em metalinguagem. Se eu me dirijo a você, leitor, e me refiro ao momento em que escrevo esse parágrafo, estou escrevendo um texto que está se referindo ao próprio texto, sendo assim eu posso dizer que isso é metalinguagem.

Dessa forma, podemos dizer que *metadados* são dados que se referem aos próprios dados. A palavra metadados é muito utilizada em computação dentro de diferentes contextos e por isso precisamos ser um pouco mais específicos em relação a que tipo de metadados estamos nos referindo. Se estamos falando de banco de dados, os dados são aqueles que estão armazenados no banco e os metadados são a descrição desses dados, em outras palavras, a estrutura das tabelas. Para um documento

XML, os metadados são representados pelo documento que descreve a estrutura do documento XML, como o DTD ou XML Schema.

Quando o contexto é uma aplicação orientada a objetos, podemos dizer que os dados são representados pelas instâncias e os metadados são os dados que descrevem essas instâncias, em outras palavras, as informações a respeito das classes. Sendo assim, os atributos, métodos, interfaces e superclasse são metadados relacionados a classe. Dessa forma, trabalhar com reflexão é na verdade trabalhar com os metadados do programa, utilizando-os para criar algoritmos que trabalham com uma classe que não conhecem previamente.

A questão é que somente os metadados da própria classe muitas vezes não são suficientes para um componente que utiliza reflexão. Podem ser necessárias mais informações para saber ao que um atributo está se referindo, quando um método precisa ser invocado e como uma determinada classe deve ser utilizada pelo componente.

Esse capítulo não irá falar sobre os metadados da própria classe, que foram vistos no capítulo anterior, mas de como definir metadados adicionais para serem utilizados para diferenciar uns elementos de um programa dos outros. Esses metadados são chamados de metadados específicos de domínio, pois eles possuem uma semântica específica para serem lidos e tratados por um componente com um objetivo específico.

Esse capítulo fala sobre esse tipo de metadados, as formas que eles podem ser definidos e as vantagens e desvantagens de cada abordagem. Adicionalmente, será dado uma ênfase nas anotações, que é um mecanismo da linguagem Java para a definição de metadados diretamente no código. Será mostrado como definir novas anotações, inseri-las nos elementos de código e consumi-las em tempo de execução através da API Reflection.

3.1 Definição de metadados

"Se alguma coisa irá variar de forma previsível, guarde a descrição dessa variação em um banco de dados de forma que seja fácil de mudar."
– Ralph Johnson

No contexto da orientação a objetos, os metadados são informações sobre os elementos do código. Essas informações podem ser definidas em qualquer meio, bastando que o software ou componente as recupere e as utilize para agregar novas informações nos elementos do código. Essa seção irá abordar diferentes formas de

definição de metadados, falando sobre as vantagens e desvantagens de cada uma delas.

Convenções de código

A primeira forma de definição de metadados que acabamos utilizando sem perceber muito quando começamos a utilizar reflexão são as convenções de código. Essa estratégia utiliza as próprias construções da linguagem para criar uma semântica particular a ser utilizada pelo componente. Em outras palavras, a presença de determinados elementos, como um padrão de nome, um tipo de retorno ou a implementação de uma interface, passam a ter um significado especial para o componente. A partir dessas informações é possível fazer a distinção desses elementos, tratando-os de forma diferente dos outros.

Os próprios exemplos que vimos nos capítulos anteriores, acabam utilizando convenções de código de alguma forma. O componente que executa as rotinas de validação, considera o prefixo `verificar` como forma de identificar os métodos de validação e a implementação da interface `Validador` para identificação dos atributos com objetos que precisam ser executados. Um outro padrão de código amplamente utilizado pelos componentes é o Java Beans [4], que define que métodos de recuperação de informações devem começar com `get` e métodos para inserir valores em propriedades devem ser começados com `set`, conforme ilustrado na próxima listagem. Essa convenção foi utilizada no exemplo da geração do mapa de propriedades e é utilizada em diversos outros componentes e frameworks.

Listagem 3.1 - Exemplo de convenção de código de um Java Bean:

```
public class JavaBean {
    private String propriedade;
    public String getPropriedade(){
        return propriedade;
    }
    public void setPropriedade(String s){
        this.propriedade = s;
    }
}
```

Um outro exemplo de convenção de código muito utilizada foi o prefixo `test` no JUnit 3. Nessa versão e nas anteriores, um método com esse prefixo, como ilustrado na próxima listagem era considerado um método de teste e invocado pelo

framework. Em versões posteriores, o uso desse prefixo foi substituído por uma anotação.

Listagem 3.2 - Definição de um teste no JUnit 3:

```
public class TesteSomador extends TestCase {

    //identificado como teste pelo prefixo "test"
    public void testSoma(){
        Somador s = new Somador();
        int resultado = s.somar(2,2,4);
        assertEquals(8,resultado);
    }
}
```

Porém não existem somente convenções que utilizam prefixos e sufixos nos nomes. Um exemplo comum é utilizar o próprio nome do elemento de código para que ele corresponda ao nome utilizado em alguma outra entidade que ele está mapeando. No exemplo do gerador de mapa de propriedades, o nome do método foi utilizado para gerar o nome da propriedade. De forma análoga, é possível mapear o nome de uma classe para o nome de uma tabela no banco de dados que ela representa ou para uma URL de uma aplicação web que ela precisa tratar.

As interfaces de marcação foram uma prática comum para definir um metadado antes de Java ter suporte nativo a anotações. A interface de marcação é uma interface sem métodos, a qual as classes implementam para terem um significado especial para algum componente. Um exemplo disso é a interface `Serializable`, que marca as classes que são serializáveis. Essa interface não possui métodos e é adicionada a classe apenas com o intuito de marcá-la como tendo uma determinada propriedade. A próxima listagem mostra uma classe válida que implementa essa interface, não precisando implementar nenhum método. Em outras palavras, essa interface serve mais para adicionar um metadado do que para realmente representar uma abstração.

Listagem 3.3 - Código válido que implementa a interface Serializable:

```
public class Serializavel implements Serializable {
    public int numero;
    public String texto;
    //sem métodos omitidos
}
```

Uma das características do uso das convenções de código é utilizar mecanismos da própria linguagem para a definição dos metadados. O lado positivo disso é que os desenvolvedores não precisam fazer nada adicionar para que o componente entenda os metadados daquela classe, bastando seguir as convenções na definição dos nomes e elementos do código. O lado ruim dessa característica é que os metadados ficam, de certa forma, implícitos e precisam ser compartilhados por toda equipe para seu uso ser efetivo. Por exemplo, pode não ser claro para um desenvolvedor que a mudança de nome de um método pode causar uma mudança de comportamento em um componente que invoca aquela classe.

Devido a facilidade de utilização de convenções de código, essa prática é bastante utilizada por frameworks e APIs, porém essa é raramente uma técnica que é utilizada sozinha. Um dos motivos é que a expressividade dessa técnica é bastante limitada e apenas metadados simples conseguem ser representados dessa forma. Sendo assim, o que costuma ser utilizado são algumas convenções de código que são complementadas com alguma das outras formas de definição de metadados. No exemplo da geração do mapa de propriedades, uma convenção de código considerava que o nome da chave do mapa seria igual ao nome utilizado no método getter, porém uma anotação permitia que um nome diferente fosse utilizado.

Definição programática

Uma outra forma de definição de metadados é através de uma rotina que insere as informações relativas aos metadados diretamente ao componente. Seria algo que como a chamada de uma método que inserisse a informação de que uma classe possui uma determinada informação, ou que algum de seus métodos possuem um dado associado. Essa abordagem tira a responsabilidade do programa de ler os metadados, pois é o programa que deve inseri-los. Em um programa que utilizar esse componente, a rotina de inserção desses metadados deve ocorrer antes que esse componente utilize uma classe, em algum código que normalmente é colocado na inicialização da aplicação.

Não existem muitos exemplos de frameworks e APIs que utilizam essa abordagem. Do ponto de vista de quem utiliza o componente, pessoalmente não acho muito intuitivo utilizar um código imperativo para configurar informações que são estáticas. Um framework brasileiro que utiliza esse tipo de técnica é o Mentawai.

Na listagem a seguir está um exemplo obtido do tutorial desse framework que faz o mapeamento de uma classe para a base de dados. Nesse exemplo, a classe `Usuario` é mapeado para a tabela `Usuarios` e seus atributos são mapeados para

as tabelas. Cada chamada do método `field()` passa como parâmetro o nome do atributo, opcionalmente o nome da coluna caso seja diferente do atributo e o tipo do campo no banco de dados. Apesar de serem chamadas de método, é possível perceber que eles estão definindo informações a respeito dos elementos da classe.

Listagem 3.4 - Exemplo de configuração programática de metadados de persistência no framework Mentawai:

```
@Override
public void loadBeans() {
    bean(Usuario.class, "Usuarios")
        .pk("id", DBTypes.AUTOINCREMENT)
        .field("login", DBTypes.STRING)
        .field("senha", DBTypes.STRING)
        .field("dataNascimento", "nascimento", DBTypes.DATE);
}
```

Apesar dessa definição programática de metadados não ser diretamente utilizada pelos desenvolvedores, ela acaba existindo internamente em diversos componentes e frameworks. Isso acontece porque se houver a funcionalidade de ler os metadados definidos de alguma outra forma, a classe que faz isso precisará dessa interface programática para poder inseri-los.

Fontes externas

Outra forma de definir os metadados é utilizando fontes externas de dados. Dessa forma, o componente ou framework deve acessar essa fonte externa de dados e realizar a leitura dessas informações. De alguma forma, elas devem referenciar os elementos de código, como classes, métodos e atributos, para que quem realizar essa leitura possa referenciar a metainformação à quem ela pertence. Essa leitura também deve ser feita antes que o componente seja utilizado, pois nesse momento essas informações já devem estar presentes.

Uma das formas mais comuns de se definir metadados em uma fonte externa é através de arquivos de configuração, sendo que arquivos XML acabam sendo os mais utilizados. Esses arquivos costumam possuir um formato bem definido para poderem ser lidos pelos pelos componentes. Apesar de XML ser considerado um formato de arquivo "amigável para pessoas", muitas vezes a configuração das informações é complicada e tediosa. O fato das informações precisarem referenciar os elementos do código faz com que o arquivo acabe ficando verboso, sendo que como são raros

os casos em que existe algum suporte de ferramenta para trabalhar com esse arquivo, é muito fácil cometer um erro em um nome de classe ou método.

Outra alternativa para armazenar os metadados são os bancos de dados. Se eles podem armazenar dados da aplicação, porque não armazenar dados sobre a aplicação. Apesar dessa não ser uma alternativa muito utilizada por frameworks de mercado devido a dificuldade de configuração inicial, em componentes caseiros, ou seja, desenvolvidos dentro da aplicação, essa acaba sendo uma alternativa bastante utilizada. Principalmente para metainformações que podem ser alteradas em tempo de execução, como permissões de segurança de métodos de negócio, ter um controle transacional e centralizado desses dados pode ser uma boa ideia.

Para exemplificar a configuração dos metadados em fontes externas ao código, a próxima listagem trás os metadados de um mapeamento objeto-relacional feito pelo framework Hibernate. Observe que o elemento `<class>` referencia a classe `com.casadocodigo.Usuario` e, em seguida, os elementos `<property>` referenciam os atributos, associando-os a respectiva coluna na base de dados. Os metadados representados aqui em XML possuem o mesmo objetivo dos definidos de forma programática na listagem anterior.

Listagem 3.5 - Exemplo de definição de metadados de persistência em arquivos XML:

```
<hibernate-mapping>
     <class name="com.casadocodigo.Usuario" table="USUARIO">
          <id column="USER_ID" name="id" type="java.lang.Long">
               <generator class="org.hibernate.id.TableHiLoGenerator">
                    <param name="table">idgen</param>
                <param name="column">NEXT</param>
               </generator>
          </id>
          <property column="LOGIN" name="login" type="java.lang.String"/>
        <property column="SENHA" name="senha" type="java.lang.String"/>
        <property column="NASCIMENTO" name="dataNascimento"
                type="java.util.Date"/>
     </class>
</hibernate-mapping>
```

Uma das desvantagens da definição de metadados em fontes externas é que os dados precisam referenciar os elementos do código tornando essa configuração tediosa e sujeita a erros, principalmente se não houver uma ferramenta de apoio. Utilizando

essa abordagem acaba havendo uma distância entre os elementos do código e os metadados. Isso causa dificuldades na sua configuração e, principalmente, quando for necessária alguma manutenção. Uma alteração inofensiva nos elementos de uma classe pode quebrar uma referência nos arquivos de configuração e gerar um bug indesejado na aplicação.

Os metadados, de uma certa forma, acabam sendo responsáveis por parte do comportamento da aplicação. Dessa forma, eles devem ser versionados junto com a aplicação para poderem gerar o comportamento desejado. Para documentos XML isso não é um grande problema, mas para bancos de dados pode ser algo bastante complicado. Principalmente quando os usuários de alguma forma podem modificar os metadados e é preciso fazer uma junção ou adaptação dos metadados devido a novas classes ou modificações em classes existentes.

Por outro lado, a definição de metadados em fontes externas é uma boa alternativa quando é preciso dar suporte a configuração de metadados para classes de terceiros e que não se tem acesso de modificação do código fonte. Nesse caso, essa desvinculação entre os metadados e as classes é uma vantagem. Adicionalmente, essa abordagem também é adequada para permitir que os metadados possam ser alterados sem a necessidade de recompilação da aplicação. Isso é ideal para quando são necessários ajustes nos metadados em tempo de deploy ou em tempo de execução.

Anotações

As anotações são um recurso da linguagem Java introduzidas na JDK 5 para permitir a adição de metadados diretamente no código. Essa prática também é conhecida também como programação orientada a atributos. A partir desse recurso é possível manter o código e os metadados no mesmo local, simplificando sua administração. Essa abordagem acaba também diminuindo a verbosidade de sua definição, visto que como os metadados são adicionados nos próprios elementos de código, não é necessário referenciá-los. Esse recurso de linguagem também está presente em outras linguagens como no C#, onde é chamado de *attributes*. A linguagem Phyton possui um mecanismo um pouco diferente chamado de *decorator*, mas que também é utilizado para a adição de metadados diretamente no código.

Na linguagem Java, a programação orientada a atributos começou antes das anotações, com a ferramenta XDoclet. No Java EE 1.4, que na época ainda recebia o nome de J2EE, a criação de EJBs demandava a criação de diversos descritores XML e interfaces. Eu cheguei a desenvolver para essa plataforma e posso afirmar que realmente era muito complicado desenvolver sem uma boa ferramenta de apoio. Nessa

época, uma alternativa que surgiu para o desenvolvimento de EJBs foi a utilização da ferramenta XDoclet.

O XDoclet é uma engine de geração de código que processa o código fonte permitindo a adição de metadados em tags JavaDoc, que são adicionadas dentro de comentários. Como resultado desse processamento é possível gerar descritores XML e o código fonte de outras classes. Existia um grande conjunto de tags, por exemplo, que eram destinadas ao desenvolvimento para a plataforma J2EE. No caso de um EJB de sessão, por exemplo, eram geradas as interfaces auxiliares e os descritores XML em que aquela classe deveria estar presente. A listagem a seguir mostra o exemplo de uma classe anotada com as tags JavaDoc do XDoclet.

Listagem 3.6 - Utilização do XDoclet para geração de EJBs:

```
/**
 * @ejb.bean
 *   name="CustomerService"
 *   jndi-name="CustomerServiceBean"
 *   type="Stateless"
**/
public abstract class CustomerServiceBean implements SessionBean {

  /**
   * @ejb.interface-method tview-type="both"
   */
  public void createCustomer(CustomerVO customer) {
  }

  /**
   * @ejb.interface-method tview-type="both"
   */
  public void updateCustomer(CustomerVO customer) {
  }
}
```

Não sei exatamente qual foi a motivação para a introdução das anotações no Java 5, mas existiam diversas forças que acabaram empurrando a linguagem nessa direção. Por um lado havia uma grande critica devido a grande quantidade de descritores do Java EE e a programação orientada a atributos com o XDoclet se mostrava como uma alternativa viável. Por outro lado, a linguagem C# já possuía o mecanismo de

atributos e este vinha sendo utilizado com sucesso para o desenvolvimento de aplicações corporativas.

As anotações foram anunciadas na linguagem Java como o recurso que iria simplificar o desenvolvimento de aplicações corporativas. Hoje é possível ver que realmente elas conseguiram cumprir essa promessa, eliminando a obrigatoriedade de diversos descritores XML que tornavam o desenvolvimento desse tipo de aplicação bastante burocrático. Atualmente, grande parte dos frameworks e APIs para a linguagem Java fazem o uso de anotações. Esse recurso de linguagem acabou popularizando a utilização de frameworks baseados em metadados, pois facilitou a definição dos metadados para os seus usuários, tornando o uso dessa abordagem mais viável.

Para exemplificar a utilização de anotações para definição de metadados, considere a listagem a seguir que mostra como o mapeamento para um banco de dados é feito com as anotações da API JPA. Observe, por exemplo, que nada precisou ser feito para os atributos `login` e `senha`, que nos exemplos anteriores precisaram ser referenciados explicitamente na definição de metadados. A anotação `@Table`, por exemplo, só precisa ser adicionada caso o nome da tabela seja diferente do nome da classe. O mesmo se aplica a anotação `@Column`, que também só é necessária quando a coluna tem o nome diferente do atributo.

Listagem 3.7 - Mapeamento objeto-relacional com JPA utilizando anotações:

```
@Entity
@Table(name="Usuarios")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

     private String login;
     private String senha;

     @Column(name="nascimento")
     private Date dataNascimento;

    //métodos get e set omitidos
}
```

Apesar de serem mais fáceis de serem configuradas por ficarem próximas aos

elementos de código, essa própria característica também pode ser uma desvantagem das anotações. Quando elas são utilizadas, para a alteração dos metadados é necessário recompilar as classes, o que pode não ser viável para ajustes em tempo de implantação ou alteração em tempo de execução. Isso também pode dificultar a reutilização da classe em um contexto em que as anotações não sejam necessárias ou que outros metadados precisem ser configurados.

O uso de anotações nas classes talvez seja familiar a grande parte dos desenvolvedores, porém a criação de novas anotações e o desenvolvimento de componentes que recuperam e fazem processamentos baseados nelas, talvez não sejam tão populares. O resto desse capítulo se dedica a mostrar como as anotações podem ser criadas e como utilizar a API Reflection para recuperá-las.

Misturando mecanismos de definição de metadados

Como pode ser visto, cada forma de definição de metadados possui suas vantagens e desvantagens, e muitas vezes fica difícil equilibrar os requisitos a partir de uma das formas. Por exemplo, enquanto as anotações oferecem uma forma menos verbosa de definir os metadados, as fontes de dados externas permitem a alteração dos metadados sem a recompilação das classes. Felizmente, os componentes não precisam escolher apenas uma forma de definição de metadados, sendo possível combinar metadados de diferentes fontes. O capítulo **??** fala como estruturar o componente para permitir a combinação de metadados de diferentes fontes.

3.2 Criando anotações

As anotações em Java são definidas em arquivos separados com extensão `.java` que precisam ter seu mesmo nome, assim como as classes, as interfaces e as enumerações. Para definir uma nova anotação, é preciso utilizar a palavra chave `@interface`. A listagem a seguir mostra um exemplo simples de como se definir uma anotação.

Listagem 3.8 - Definição de uma simples anotação chamada Metadado:

```
public @interface Metadado {
}
```

Uma anotação pode possuir atributos, ou seja, você pode agregar informações aos metadados, porém não é qualquer classe que é aceita como tipo do atributo. Porém uma anotação não pode possuir comportamento, sendo uma informação estática que é adicionada a classes. Existem também anotações que podem ser adicionadas as próprias anotações para parametrizar como elas devem ser tratadas pelo compilador e pela máquina virtual. As próximas subseções irão apresentar detalhes a respeito de como são criadas as anotações.

Uma coisa importante das anotações é: **elas não fazem nada!!!** Para quem utiliza um framework que utiliza anotações, pode parecer que a configuração de anotações adiciona comportamentos na classe, porém a verdade é que elas somente estão agregando novas informações sobre os elementos da classe. Para que elas tenham algum efeito sobre o funcionamento do programa, alguém precisa recuperar essas anotações e fazer alguma coisa a respeito, pois sozinhas as anotações não fazem nada.

Definindo até quando a anotação está disponível

Uma das principais configurações de uma anotação é até quando ela vai estar disponível para recuperação. Dependendo do uso que será feito dela e do momento em que essa informação será consumida, diferentes tipos de retenção podem ser necessárias. Segue uma lista com os três diferentes tipos de retenção que uma anotação pode possuir:

- `SOURCE` : Uma anotação com esse tipo de retenção fica disponível apenas no código fonte. No momento em que a classe é compilada, a anotação não é transferida para o arquivo `.class`. Esse tipo de anotação é normalmente utilizada para fins de documentação e para uso de ferramentas que fazem processamento direto de código fonte. Um exemplo são ferramentas que fazem validações em tempo de compilação.

- `CLASS`: Anotações com esse tipo de retenção são mantidas nos arquivos `.class`, porém não são carregadas pela máquina virtual. Nesse caso, elas ficam disponíveis até o momento do carregamento da classe. Esse tipo de anotação é utilizada por ferramentas que fazem processamento do bytecode da classe, podendo esse processamento ser feito de forma estática como uma etapa posterior a compilação, ou no momento do carregamento das classes. Esse tipo de prática será visto no capítulo **??**.

- `RUNTIME`: Para uma anotação estar disponível para a recuperação em tempo de execução, ela precisa ter esse tipo de retenção. Nesse caso, a máquina virtual vai carregar essa anotação em memória e torná-la acessível através da API Reflection. Esse tipo de anotação é o tipo utilizado por frameworks que precisam ter acesso as anotações em tempo de execução. Será o tipo que iremos utilizar com maior frequência aqui nesse livro.

Para configurar o tipo de retenção de uma anotação é preciso anotá-la com a anotação `@Retention`. Essa anotação recebe como parâmetro uma enumeração do tipo `RetentionPolicy`, que pode possuir os valores `SOURCE`, `CLASS` ou `RUNTIME` descritos acima. A listagem a seguir exemplifica como realizar essa configuração. Nesse capítulo iremos abordar somente as anotações com retenção do tipo `RUNTIME`, ficando as anotações do tipo `CLASS` para o capítulo **??**. O processamento de anotações diretamente no código fonte como parte do processo de compilação, que utilizaria anotações do tipo `SOURCE`, estão fora do escopo desse livro.

Listagem 3.9 - Configurando o tipo de retenção de uma anotação:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Metadado {
}
```

Caso uma anotação não possua uma retenção configurada, por definição ela será do tipo `SOURCE`. Sendo assim, se você estiver desenvolvendo um código que consome uma anotação e por algum motivo essa anotação não estiver sendo encontrada, muito provavelmente é porque você esqueceu de configurar a retenção para `RUNTIME`. Eu já criei diversos frameworks que consomem anotações, e mesmo assim de tempos em tempo isso acaba acontecendo comigo.

Tipo de elementos anotados

Outra configuração importante para uma anotação são os tipos de elementos que podem ser anotados por ela. Se nada for dito, a anotação poderá ser adicionada em qualquer tipo de elemento. Por mais que essa configuração não seja obrigatória, é sempre uma boa prática adicioná-la para evitar que os seus usuários adicionem-na no local errado. Por exemplo, uma anotação que deve ser adicionada nos métodos de acesso pode ser adicionada por engano em um atributo, o que vai fazer que não seja encontrada pela classe que a procurar.

Segue abaixo uma lista com os tipos de elemento que podem ser anotados, sendo cada um deles um elemento da enumeração `ElementType`:

- `TYPE`: qualquer definição de tipo, como classes, interfaces e enumerações.
- `PACKAGE`: pacotes.
- `CONSTRUCTOR`: construtores.
- `FIELD`: atributos.
- `METHOD`: métodos.
- `PARAMETER`: parâmetros de métodos e construtores.
- `LOCAL_VARIABLE`: variáveis locais.
- `ANNOTATION_TYPE`: anotações.

Para configurar o tipo de elemento que uma anotação pode anotar, é preciso configurá-la com a anotação `@Target`. Essa anotação pode receber um array com diversos tipos onde ela faz sentido ser adicionada. A listagem a seguir mostra um exemplo de configuração, onde a anotação pode ser adicionada em um método ou em um atributo.

Listagem 3.10 - Configurando o tipo de elemento que pode ser anotado:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
public @interface Metadado {
}
```

Em relação ao tipo `LOCAL_VARIABLE`, como as variáveis locais não são acessíveis por reflexão, esse tipo de anotação só faz sentido com retenção do tipo `SOURCE` ou `CLASS`. Outro tipo de anotação curioso, é a anotação do tipo `PACKAGE` que é utilizada para adicionar metainformações em uma pacote. Para adicioná-la, é preciso criar um arquivo no pacote chamado `package-info.java` para definir a anotação. A listagem a seguir mostra como seria o conteúdo desse arquivo.

Listagem 3.11 - Conteúdo do arquivo package-info.java para adicionar anotação ao pacote:

```
@Metadado
package org.casadocodigo;

import org.anotacoes.Metadado;
```

Outras definições

Além das anotações `@Target` e `@Retention` existem outras que podem ser adicionadas nas anotações para configurar como as ferramentas da JDK devem encará-las. Uma delas é a anotação `@Documented` que deve ser utilizada quando a anotação precisar ser incorporada a documentação dos elementos anotados. Por exemplo, uma anotação que determina os valores válidos para um atributo, certamente é interessante de ser adicionada na documentação. Outras anotações muito específicas, que por exemplo marca um método para ter suas invocações incluídas no log, talvez não precise entrar na documentação.

Outra anotação que pode ser adicionada na anotação é `@Inherited`. Nesse caso, a anotação será propagada para as subclasses da classe que a possuir. Uma nota importante é que isso só irá funcionar com anotações com `@Target` do tipo `TYPE`. Ao contrário do que se esperaria, um método sobrescrito por uma subclasse não irá herdar as anotações do tipo `@Inherited` do método que está sobrescrevendo.

Atributos da anotação

Em alguns casos, apenas a presença da anotação já é suficiente para configurar uma propriedade do elemento anotado, como se a classe é persistente ou não. Porém em outros casos, são necessárias informações adicionais, como, por exemplo, qual o nome da tabela do banco de dados para a qual essa classe é mapeada. Sendo assim, as anotações podem possuir atributos para a configuração de valores que fazem parte do metadado. Diferentemente de um atributo de classe, apenas alguns tipos podem ser tipos de atributos de uma anotação. Esses são tipos primitivos, enums, `Class`, `String`, outras anotações e arrays de qualquer um desses tipos.

Todo atributo de uma anotação precisa possuir um valor. Dessa forma, se o atributo estiver presente, ele precisa de um valor. É possível configurar um valor *default* para o atributo e quando isso é feito, aquele valor não precisa obrigatoriamente ter um valor atribuído quando a anotação é utilizada. Os atributos são configurados através de pares nome e valor adicionados na anotação no seguinte formato: `@Anotacao(nome=valor)`. Existe um nome especial de atributo, `value`, que caso seja o único atributo configurado da anotação pode ter seu nome omitido da seguinte

forma: `@Anotacao(valor)`.

Para entender melhor o que é válido e o que não é, nada melhor do que ver alguns exemplos. A listagem a seguir, mostra uma anotação com um atributo chamado `value`. Ao definir a anotação sem o atributo em `atributo1`, isso é inválido pois o valor precisa ser definido obrigatoriamente. Como o atributo da anotação se chama `value`, então ele pode ser explicitamente referenciado ou não, respectivamente na forma `@Metadado(value=13)` ou `@Metadado(13)`, sendo as duas formas válidas.

Listagem 3.12 - Exemplo de anotação com atributo value:

```
//definição da anotação
public @interface Metadado {
    int value();
}

//Exemplos de uso
public class Exemplo{

    @Metadado //inválido pois não definiu atributo
    public String atributo1;

    @Metadado(value=13) //válido
    public String atributo2;

    @Metadado(13) //válido
    public String atributo3;
}
```

O exemplo apresentado na próxima listagem apresenta uma anotação com o atributo `tipo`, que possui a definição de um valor *default* igual a uma `String` vazia. Nesse caso, a definição da anotação da forma `@Metadado`, sem definição de um valor para o atributo, é válida por causa de um valor ter sido definido por definição na própria anotação. Para essa anotação, a omissão do nome do atributo é inválida quando ele é configurado, pois o nome do atributo não é `value`.

Listagem 3.13 - Exemplo de anotação com atributo value e valor default:

```
//definição da anotação
public @interface Metadado {
```

```
    String tipo() default "";
}

//Exemplos de uso
public class Exemplo{

    @Metadado //válido
    public String atributo1;

    @Metadado(tipo="OK") //válido
    public String atributo2;

    @Metadado("OK") //inválido porque o atributo não chama value
    public String atributo3;
}
```

Como último exemplo de definição de atributos, a próxima listagem mostra uma anotação com um atributo `value` e um atributo chamado `tipo` com valor *default*. A anotação em `atributo1` é inválida pois o atributo `value` precisa ter um valor definido e a anotação em `atributo2` é válida pois quando somente o atributo `value` é definido, seu nome pode ser omitido. Nos dois exemplos de uso seguintes, podemos ver que o atributo `value` só pode ser omitido quando é o único atributo definido na anotação. Sendo assim, quando o atributo `tipo` é definido, o nome do atributo `value` precisa ser definido explicitamente.

Listagem 3.14 - Exemplo de anotação com atributo value e valor default:

```
//definição da anotação
public @interface Metadado {
    Class value();
    String tipo() default "";
}

//Exemplos de uso
public class Exemplo{

    @Metadado //inválido porque value não tem valor default
    public String atributo1;

    @Metadado(String.class) //válido
    public String atributo2;
```

```
    @Metadado(value=String.class, tipo="OK") //válido
    public String atributo3;

    @Metadado(String.class, tipo="OK")
    //inválido pois value só pode ser omitido quando é o único atributo
    public String atributo4;
}
```

3.3 Lendo anotações em tempo de execução

Depois de ver na seção anterior como criar anotações, essa seção mostra como recuperá-las através da API Reflection. Mais uma vez ressalto que para isso ser possível é preciso que a anotação possua a definição `@Retention` com o valor `RUNTIME`, pois caso contrário a máquina virtual não irá carregar a definição do metadado junto com a classe.

Recuperando anotações

Existe uma interface chamada `AnnotatedElement` que define os métodos para a recuperação de anotações. Essa interface é implementada por todas as classes que representam elementos de código que podem ser anotados, como as classes `Class`, `Method`, `Field`, `Package` e `Constructor`. Sendo assim, esses métodos que serão apresentados podem ser invocados em qualquer uma dessas classes.

Existem dois métodos nessa interface que nos permite trabalhar com uma anotação específica. O primeiro deles é `isAnnotationPresent()`, que pode ser utilizado para verificar se uma determinada anotação está presente ou não no elemento. Esse método retorna um valor booleano dizendo se a anotação existe naquele elemento. O outro método é `getAnnotation()`, que recebe como parâmetro a classe da anotação e retorna a anotação desejada. A instância retornada possui o tipo da anotação, e seus atributos podem ser recuperados através dos métodos com seus nomes.

A listagem a seguir mostra um exemplo de código onde uma anotação é recuperada e seus atributos impressos no console. Antes de imprimir a anotação, o método `isAnnotationPresent()` é utilizado para confirmar a presença da anotação. Observe que quando o método `getAnnotation()` é invocado, ele recebe a classe da anotação como parâmetro e utiliza seu tipo genérico para inferir o tipo do retorno, que é o próprio tipo da anotação. A partir dessa instância com

o tipo da anotação, é possível obter os seus atributos invocando métodos com seus nome, conforme mostrado no exemplo.

Listagem 3.15 - Recuperação da anotação:

```
//definição da anotação
@Retention(RetentionPolicy.RUNTIME)
public @interface Metadado {
    String nome();
    int numero();
}

//Uso e recuperação de anotação
@Metadado(nome="classe", numero=17)
public class RecuperaAnotacao {

    public static void main(String[] args) {
        Class<RecuperaAnotacao> c = RecuperaAnotacao.class;
        if(c.isAnnotationPresent(Metadado.class)){
            Metadado m = c.getAnnotation(Metadado.class);
            System.out.println("Propriedade nome = "+ m.nome());
            System.out.println("Propriedade numero ="+ m.numero());
        }
    }
}
```

Os outros métodos definidos em `AnnotatedElement` para recuperação de anotações retornam a lista de anotações de um elemento. O método `getAnnotations()` irá retornar uma lista com todas as anotações de um determinado elemento de código e o método `getDeclaredAnnotations()` irá retornar somente as anotações declaradas no elemento, excluindo as com `@Inherited` que foram herdadas da superclasse. Como a herança de anotações funciona somente para classes, para outros tipos de elemento o retorno dos dois métodos será sempre o mesmo.

Anotações em parâmetros

As anotações em parâmetros são recuperadas de forma diferente, pois não existe uma classe na API Reflection que representa um parâmetro. O método para a recuperação de anotações em parâmetros se chama `getParameterAnnotations()` e

está presente nas classes `Method` e `Constructor`. Essa chamada retorna um array de duas dimensões de `Annotation`, sendo que a primeira dimensão representam os parâmetros e a segunda dimensão representa as anotações daquele parâmetro.

As anotações em parâmetros são muito úteis para identificar qual informação precisa ser passada para ele. Dessa forma, quando um método for ser invocado utilizando reflexão, essa informação pode ser utilizada para descobrir que informação se espera receber. Nos exemplos que foram vistos até agora, eram utilizadas convenções para determinar quais seriam os parâmetros dos métodos invocados, porém utilizando essas informações é possível lidar com diversas combinações de parâmetros, reconhecendo-os em tempo de execução.

Para exemplificar esse procedimento, a próxima listagem apresenta a definição de uma anotação de parâmetros. Essa anotação se chama `@Param` e o objetivo é que um nome para o parâmetro seja definido em seu atributo `value`.

Listagem 3.16 - Anotação para nomear os parâmetros:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface Param {
    String value();
}
```

A próxima listagem está um método chamado `invocarMetodo()` que recebe um `Method`, um objeto e um mapa de informações como parâmetro, e executa o método recuperando os seus argumentos do mapa de acordo com o nome configurado na anotação. Observe que esse método inicialmente recupera as anotações dos parâmetros pelo método `getParameterAnnotations()` e em seguida cria um array chamado `paramValues` para armazenar os valores a serem passados na chamada do método. Para cada parâmetro, o método auxiliar `getNomeParâmetro()` recebe a lista de anotações daquele parâmetro e busca entre elas a anotação `@Param`, retornando o seu valor. Em seguida, é recuperado do mapa o objeto com a chave tendo o mesmo nome configurado na anotação. Após recuperar todos os parâmetros, o método é invocado passando o array resultante.

Listagem 3.17 - Invocação de método recuperando os parâmetros de um mapa:

```
public static Object invocarMetodo(Method m, Object obj,
        Map<String,Object> info) throws Exception{
```

```
    Annotation[][] paramAnnot = m.getParameterAnnotations();
    Object[] paramValues = new Object[paramAnnot.length];
    for(int i=0; i<paramValues.length; i++){
        String name = getNomeParametro(paramAnnot[i]);
        paramValues[i] = info.get(name);
    }
    return m.invoke(obj, paramValues);
}
public static String getNomeParametro(Annotation[] ans){
    for(Annotation a : ans){
        if(a instanceof Param){
            return ((Param)a).value();
        }
    }
    throw new RuntimeException("Anotação @Param não encontrada");
}
```

A listagem a seguir mostra um exemplo de uso desse método. Um método chamado `metodo` com dois parâmetros anotados foi criado para testarmos a invocação. No método `main()`, inicialmente é criado um mapa com diversos valores para a recuperação dos parâmetros. Em seguida, um referência para o método `metodo` é recuperada e o método `invocarMetodo()` é invocado passando esse método, a instância da classe e o mapa.

Listagem 3.18 - Testando a invocação de métodos com parâmetros nomeados:

```
public class AnotacaoParametro {

    public static void main(String[] args) throws Exception {
        Map<String,Object> info = new HashMap<>();
        info.put("inteiro", 13);
        info.put("numero", 23);
        info.put("string", "OK");
        info.put("texto", "NOK");

        AnotacaoParametro ap = new AnotacaoParametro();
        Method m = ap.getClass().getMethod(
            "metodo", Integer.class, String.class);
        invocarMetodo(m, ap, info);
    }
    public void metodo(
```

```
            @Param("inteiro") Integer i, @Param("texto") String s){
        System.out.println("Parametro inteiro = "+i);
        System.out.println("Parametro texto = "+s);
    }
}
```

Deixo como observação desse exemplo que muitas coisas podem dar errado caso o tipo do objeto recuperado do mapa não seja o mesmo do esperado pelo parâmetro. Deixo como exercício ao leitor complementar esse exemplo, verificando se os tipos são compatíveis e fazendo uma conversão quando possível. Como saber como a conversão deve ser feita? Tente utilizar uma anotação no próprio parâmetro para indicar métodos ou classes que devem ser utilizados para fazer essa conversão.

Lendo atributos de uma anotação com reflexão

Da mesma forma que uma classe pode ser lida através de reflexão sem a conhecermos previamente, também é possível obtermos a classe relativa a anotação para descobrirmos seus atributos em tempo de compilação. A principal diferença é que para obtermos instância de `Class` que representa a anotação, devemos chamar o método `annotationType()` ao invés de `getClass()`. Em seguida, os atributos da anotação podem ser recuperados através dos métodos que são utilizados para retorná-los.

Para exemplificar esse processo, a listagem a seguir mostra um código que recupera as anotações de um elemento e imprime no console com seus atributos. Observe que o método recebe um `AnnotatedElement` como parâmetro, permitindo que receba qualquer classe que represente um elemento que possa ter anotações. Ao recuperar a instância de `Class` referente a anotação utilizando `annotationType()`, o método imprime o nome da anotação e em seguida itera pelos seus métodos imprimindo o nome e o valor retornado. Observe que utilizando `getDeclaredMethods()` recuperamos somente os métodos declarados na anotação, ou seja, os que definem atributos.

Listagem 3.19 - Método que imprime os atributos das anotações de um elemento:

```
public static void imprimeAnotacoes(AnnotatedElement ae)
                                            throws Exception{
    Annotation[] ans = ae.getAnnotations();
    for(Annotation a : ans){
        Class<?> c = a.annotationType();
```

```
        System.out.println("@"+c.getName());
        for(Method m : c.getDeclaredMethods()){
            Object o = m.invoke(a);
            System.out.println(" |->"+m.getName()+"="+o);
        }
    }
}
```

A listagem a seguir mostra um exemplo de uma classe com duas anotações, para a qual a respectiva instância de `Class` é passada para o método `imprimeAnotacoes()`. Em seguida é mostrado a saída do console gerada por esse programa, onde as duas anotações e seus atributos são impressos.

Listagem 3.20 - Anotação para nomear os parâmetros:

```
@Metadado(nome="teste",numero=34)
@Anotacao(String.class)
public class ImprimeAnotacoes {

    public static void main(String[] args) throws Exception {
        imprimeAnotacoes(ImprimeAnotacoes.class);
    }
}
```

Listagem 3.21 - Saída do programa que imprime as anotações:

```
@Metadado
 |->nome=teste
 |->numero=34
@Anotacao
 |->value=class java.lang.String
```

3.4 Limitações das anotações

"Aceite essas limitações e se submeta a elas em vez de continuar insistindo em fazer a sua vontade."
– Emerson natal

Durante esse capítulo muita coisa foi dita a respeito do que se pode fazer com as anotações. Essa seção fala um pouco sobre as limitações dessa funcionalidade

da linguagem, falando o que não se consegue fazer. No capítulo **??** serão abordadas algumas técnicas que são frequentemente utilizadas para contornar essas limitações.

A primeira limitação é que só pode haver uma anotação de um determinado tipo para um elemento. Imagine, por exemplo, que uma anotação aponte uma classe para fazer uma validação de seus dados. A partir dessa anotação não seria possível configurar duas classes para fazer a validação, pois somente uma anotação daquele tipo pode ser adicionada. Para que isso seja possível, uma alternativa seria a própria anotação dar suporte a configuração de diversas classes. Outra alternativa frequentemente utilizada, é utilizar uma outra anotação que possua como `value` um array da anotação original, permitindo agregar diversas anotações do tipo anterior.

Outra limitação das anotações é que elas não possuem nenhuma mecanismo que permita generalizá-las, como herança ou algum outro tipo de abstração. Com esse mecanismo, seria possível, por exemplo, ao buscar uma anotação, procurar por todas que possuem uma determinada abstração. Porém, o principal ponto onde isso prejudica, é que não é possível estender as anotações para adicionar novas semânticas e novas informações.

Por exemplo, imagine que uma anotação `@A` possua uma atributo do tipo de uma anotação `@B`. Se a herança fosse possível, esse tipo `@B` poderia ser estendido por outras anotações que poderiam ser adicionadas no lugar dele. Isso permitiria um mecanismo de metadados extensível, onde novas informações poderiam ser agregadas, como em um documento XML, por exemplo. Como isso não é possível, fica-se restrito as informações presentes na anotação `@B`, não sendo possível qualquer tipo de extensão.

3.5 Mapeando parâmetros de linha de comando para uma classe

O objetivo dessa seção é mostrar um exemplo completo e realista envolvendo anotações. Um tipo de aplicação comum dos frameworks baseados em metadados são para realizar o mapeamento entre diferentes representações de uma entidade da aplicação. Como foi mostrado no início desse capítulo, existem diversos frameworks que utilizam diferentes abordagens para mapear classes da aplicação para tabelas do banco de dados. De forma similar, existem outros frameworks que mapeiam classes para entidades de documentos XML, mapeiam métodos para web services e, até mesmo, mapeiam classes de diferentes aplicações que representam o mesmo conceito.

Seguindo esse tipo de aplicação, o exemplo que será desenvolvido irá mapear os parâmetros recebido para aplicação na linha de comando para uma classe com propriedades para facilitar sua recuperação posteriormente. Os parâmetros de linha de comando são recebidos pelo método `main()` com um array de `String` na ordem que são passados para aplicação. Usualmente, utiliza-se letras na forma `-x` para indicar que a próxima palavra será o valor daquela propriedade. O exemplo que será desenvolvido irá receber como parâmetro o array de `String` com os parâmetros e irá inserir os dados em uma instância, de acordo com as anotações de sua classe, que indicarão que atributo deve receber qual parâmetro.

A próxima listagem mostra a anotação que será utilizada para mapearmos os atributos para os parâmetros. Observe que como toda anotação que precisa ser consumida em tempo de execução, ela possui a retenção como `RUNTIME`. Outro ponto importante é que essa anotação é para ser utilizada em atributos, portanto é nesses elementos que o mapeamento deve ser feito. Apesar do mapeamento ser feito nos atributos, como esses normalmente são privados por questões de encapsulamento, serão os respectivos métodos setters que devem ser utilizados para a inserção do parâmetro na classe.

Listagem 3.22 - Anotação para mapear parâmetros para atributos:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Parametro {
    String value();
}
```

Para explicar melhor como o mapeamento será feito, a classe a seguir mostra o exemplo de uma classe mapeada. O componente desenvolvido dará suporte a três tipos de parâmetros: booleanos, `String` e array de `String`. No caso de parâmetros booleanos, a presença ou não de sua marcação indica se ele é verdadeiro ou falso. Por exemplo, se `-p` estiver presente entre os parâmetros, precisa-se atribuir o valor verdadeiro no atributo `possui`. No caso de uma `String`, deve-se passar a marcação do parâmetro seguido de seu valor, como, por exemplo, `-s console` para configurar o valor do atributo `saída`. O funcionamento será similar para arrays de `String`, exceto pelo fato de diversos valores poderem ser passados depois da marcação do parâmetro.

Listagem 3.23 - Exemplo de uma classe anotada para receber os parâmetros:

```
public class ParamApp {

    @Parametro("-p")
    private boolean possui;

    @Parametro("-n")
    private boolean naoPossui;

    @Parametro("-a")
    private String[] arquivos;

    @Parametro("-s")
    private String saida;

    //métodos get e set omitidos
}
```

Para iniciar o desenvolvimento desse componente de leitura de parâmetros, vamos começar desenvolvendo o seu construtor, que irá receber a classe mapeada com a anotação `@Parametro` e já vai efetuar a leitura dos seus metadados. Como resultado, será gerado um mapa cuja chave é a marcação do parâmetro, que é passado como valor da anotação, e a respectiva instância de `Field`. Observe que a classe `MapeamentoParametros` possui um tipo genérico, que deve ser o mesmo da classe passada como parâmetro. Esse parâmetro genérico será utilizado para determinar o retorno do método que mapear os parâmetros para uma instância dessa classe.

Listagem 3.24 - Exemplo de uma classe anotada para receber os parâmetros:

```
public class MapeamentoParametros<E> {

    private Map<String, Field> parametros;
    private Class<E> clazz;

    public MapeamentoParametros(Class<E> c) {
        parametros = new HashMap<>();
        clazz = c;
        Class<?> current = c;
        while (current != Object.class) {
            for (Field f : current.getDeclaredFields()) {
```

```
                if (f.isAnnotationPresent(Parametro.class)) {
                    Parametro p = f.getAnnotation(Parametro.class);
                    parametros.put(p.value(), f);
                }
            }
            current = current.getSuperclass();
        }
    }
}
```

Segundo os requisitos, é desejado que todos os atributos sejam pesquisados, incluindo os que são privados e os da superclasse. Dessa forma, dois laços são utilizados para percorrer os atributos, sendo que o primeiro itera da classe passada como parâmetro seguindo pelas suas superclasses até `Object`, e o segundo itera sobre os atributos de cada classe recuperados com `getDeclaredFields()`.

Seguindo uma abordagem *bottom-up* de desenvolvimento, na qual primeiro se desenvolve as funções auxiliares e depois a função principal, vamos começar por desenvolver um método que retorna a instância de `Method` que representa o método setter do `Field` passado como parâmetro. Como pode ser visto no código da próxima listagem, inicialmente é gerado o nome do método setter a partir do nome do atributo e em seguida percorre-se toda a lista de métodos da classe buscando um método com esse nome. Como não se sabe exatamente qual será o tipo do parâmetro, não é possível buscar o método diretamente. Caso o método não seja encontrado, uma exceção será lançada.

Listagem 3.25 - Método que recupera o setter referente a um atributo:

```
private Method getSetter(Field f) {
    String nomeMetodo = "set"
            + Character.toUpperCase(f.getName().charAt(0))
            + f.getName().substring(1);
    for(Method m : clazz.getMethods()){
        if(m.getName().equals(nomeMetodo))
            return m;
    }
    throw new MapeamentoException(nomeMetodo +"() não encontrado");
}
```

Um outro método auxiliar a ser desenvolvido é que insere na classe o valor do parâmetro. Ele recebe a instância da classe mapeada, o nome do parâmetro e a

lista de valores que foram passados depois dele. Como pode ser visto na listagem a seguir, o método `inserir()` inicialmente recupera o `Field` relacionado com o nome do parâmetro e em seguida o método setter relacionado ao `Field` através do método `getSetter()`. Por fim, para que seja possível invocar o método, o método `recuperarValor()` retorna o valor a ser utilizado como parâmetro de acordo com o tipo esperado pelo método.

Listagem 3.26 - Método que insere um parâmetro na classe:

```
private void inserir(E instancia, String nomeParametro,
                    List<String> valores) {
    Field f = parametros.get(nomeParametro);
    if(f == null){
        throw new MapeamentoException(nomeParametro +
            ": parâmetro não previsto");
    }
    Method m = getSetter(f);
    Object valor  = recuperarValor(nomeParametro, valores, m);
    try {
        m.invoke(instancia, valor);
    } catch (IllegalAccessException | IllegalArgumentException
            | InvocationTargetException e) {
        throw new MapeamentoException(
                    "Problemas ao invocar "+m.getName(), e);
    }
}
```

A listagem a seguir mostra a recuperação do valor da lista de `String` de acordo com o tipo do parâmetro esperado pelo método. Diferentemente dos exemplos anteriores, esse se preocupa em validar se a entrada está de acordo com o esperado. Nesse caso, o número de parâmetros na lista, que são passados depois do marcador, deve ser compatível com o tipo do parâmetro. Sendo assim, se o tipo for booleano não deve haver nenhum parâmetro, se for `String` deve haver um valor, e se for um array de `String` pode haver qualquer número. Observe que o tipo considerado é o do parâmetro recebido pelo método setter e não o do atributo. Sendo assim, o atributo poderia ter um tipo diferente e ser convertido a partir do valor recebido pelo método de atribuição.

Listagem 3.27 - Método que retorna o valor de acordo com o parâmetro do

método setter:

```
private Object recuperarValor(String nomeParametro,
        List<String> valores, Method m) {
    if(m.getParameterTypes()[0] == boolean.class ||
            m.getParameterTypes()[0] == Boolean.class){
        if(valores.size() > 0){
            throw new MapeamentoException(nomeParametro +
                " não pode possuir valor");
        }else {
            return true;
        }
    }else if(m.getParameterTypes()[0].isArray()){
        return valores.toArray(new String[valores.size()]);
    }else{
        if(valores.size() != 1){
            throw new MapeamentoException(nomeParametro +
                " só pode possuir um valor");
        }else {
            return valores.get(0);
        }
    }
}
```

Para finalizar o exemplo, a listagem a seguir mostra o método `mapear()`, que é o método principal para a realização do mapeamento. Inicialmente, o método cria uma instância da classe para inserir os parâmetros, assumindo que ela possui um construtor vazio. Em seguida percorre o array de `String` recebido como parâmetro buscando por marcadores, que seriam começados com " –". Ao encontrar um marcador, os próximos valores serão inseridos na lista de valores até que um novo marcador seja encontrado ou até que se chegue ao final do array. Ao finalizar de guardar os valores relacionados com um marcador, o método `inserir()`, mostrado anteriormente, é invocado para inserir o valor na classe corretamente.

Listagem 3.28 - Método que recebe array de String e retorna instância de classe mapeada:

```
public E mapear(String[] args) {
    String nomeParametro = null;
    List<String> valores = new ArrayList<>();
```

```
    E instancia = null;
    try {
        instancia = clazz.newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
        throw new MapeamentoException(clazz.getName()
                + " não pode ser instanciada", e);
    }
    for (int i = 0; i < args.length; i++) {
        String token = args[i];
        if (token.startsWith("-")) {
            nomeParametro = token;
        } else {
            valores.add(token);
        }
        if (args.length == i + 1 || args[i + 1].startsWith("-")) {
            inserir(instancia, nomeParametro, valores);
            nomeParametro = null;
            valores.clear();
        }
    }
    return instancia;
}
```

Observe que o código criado nesse exemplo poderia ser utilizado em programas reais de linha de comando para a recuperação e interpretação de seus parâmetros. Sem a configuração de metadados adicionais aos da classe não seria possível implementar essa funcionalidade, pois as informações da própria classe não seriam suficientes para saber que parâmetro é relacionado com cada atributo. Versões mais sofisticadas poderiam prover suporte a parâmetros de outros tipos, ou mesmo prover anotações que configuram como a `String` recebida seria convertida para o tipo desejado. Esse incremento fica como exercício para os leitores.

3.6 Considerações finais

Esse capítulo falou sobre a configuração de metadados adicionais aos da classe para serem consumidos e utilizados por algoritmos que utilizam reflexão. Inicialmente, o capítulo falou sobre as opções existentes para definição de metadados, ressaltando as vantagens e desvantagens de cada uma. O uso de convenções de código é fácil de ser utilizado, mas possui expressividade limitada e nem sempre é suficiente. A definição

programática de metadados permite uma maior flexibilidade, mas acaba não sendo produtiva por utilizar código imperativo para uma configuração declarativa. Fontes externas de metadados podem ser mais facilmente modificadas, mas a definição de metadados é mais verbosa e fica distante do código. Por fim, as anotações minimizam a quantidade de configurações, mas complica o reúso da classe em outros contextos e não podem ser utilizadas para classes que não se tem acesso ao código.

Em seguida, o capítulo focou em mostrar como as anotações são definidas e utilizadas em Java. Inicialmente foi mostrado como uma nova anotação pode ser criada e quais são as opções que podem ser configuradas. Em seguida foi mostrado como as anotações podem ser recuperadas pela API de reflexão para serem utilizadas por uma classe ou componente. Por fim, um exemplo mais completo mostrou como o mapeamento entre representações de uma entidade pode ser feito através da configuração de metadados.

Capítulo 4

Proxy Dinâmico

"Nunca faça por um intermediário o que você pode fazer você mesmo"
– provérbio italiano

Nos capítulos anteriores foram mostradas as principais funcionalidades da API de reflexão. Inicialmente foi mostrado como obter a referência para as classes e como utilizar suas informações para manipular suas instâncias. Em seguida, falou-se sobre a definição de metadados e como eles podem ser utilizados para adicionar novas informações sobre os elementos de um programa. A partir dessas funcionalidades mostrou-se alguns exemplos mais completo de como essas funcionalidades podem ser utilizadas para gerar componentes reutilizáveis que poderiam ser utilizados em softwares reais.

Uma das funcionalidades mais impressionantes da API Reflection da linguagem Java é a criação de proxies dinâmicos. De uma forma bem informal, a partir deles é possível criar uma instância que implementa uma interface em tempo de execução. Em outras palavras, ele pode implementar a interface que você quiser, tendo uma forma de tratar as chamadas de método feitas para os métodos dessa interface. A

partir disso é possível ter uma única classe que pode encapsular outras classes com diferentes métodos e interfaces.

Me lembro que a minha reação ao ver o uso de um proxy dinâmico pela primeira vez foi algo como: "*Mas pode fazer isso? Vale fazer isso?*". Era como descobrir uma nova regra no meio do jogo, que certamente poderia me ajudar a vencê-lo de forma bem mais fácil e mais inteligente. A partir de alguns exemplos, esse capítulo irá mostrar, não somente como utilizar essa técnica, mas também como ela pode ser usada para introduzir novos comportamentos nas classes de forma transparente. É essa a "*magia*" que alguns frameworks utilizam para que certas coisas simplesmente aconteçam ao invocarmos um método das classes da própria aplicação.

Além do uso da API Reflection para gerar proxies dinâmicos a partir de interfaces, esse capítulo também vai mostrar o uso da ferramenta CGLib para gerar proxies dinâmicos a partir de classes, o que é muito útil para classes como Java beans. Ele também irá falar sobre o uso de anotações para auxiliar no desenvolvimento da funcionalidade do proxy dinâmico e como essa funcionalidade pode ser utilizada para gerar em tempo de execução um implementação de uma interface, sem haver nenhuma classe compilada realmente que a implemente. Como os capítulos anteriores, ao final será apresentado um exemplo mais completo que consolida o conhecimento em proxies dinâmicos, relacionando-os com o que vimos nos capítulos anteriores.

4.1 O que é um proxy?

Antes de entrarmos no detalhe de como funciona um proxy dinâmico, primeiro vamos revisar o conceito de proxy. O Proxy é um padrão de projeto que tem o objetivo de proteger o acesso de objetos, de forma transparente as classes que o utilizam. O padrão Decorator, apesar de possuir um objetivo diferente, que é de adicionar novas funcionalidades a classe, possui uma estrutura muito similar. Ambos os padrões encapsulam a classe original, intermediando o acesso a ela. Dessa forma, enquanto o controle estiver com a classe intermediária, ela pode executar verificações de segurança ou outras funcionalidades, antes de repassar o controle da execução para a classe original.

A palavra *proxy* em inglês significa *intermediário*, o que é uma metáfora adequada para esse padrão, visto que o proxy intermedia o acesso entre o objeto cliente e o objeto original. Muitas pessoas estão acostumadas com o uso do nome *proxy* para os proxies de rede, que de certa forma possuem um comportamento análogo. Nesse caso, quando existe um proxy, ao invés de enviar os pacotes de rede para o endereço

de destino, eles são enviados para o proxy, que então redireciona para o endereço original servindo como um intermediário. Tanto para o cliente, quanto para o destino original, é transparente a existência do proxy. Porém apesar dessa transparência, o proxy pode prestar diversos serviços estando no meio do caminho, como a filtragem de conteúdos, controle do uso de banda, autenticação, entre outros.

A figura 4.1 ilustra o funcionamento de um proxy. O primeiro ponto importante é que o objeto que faz o papel do proxy deve possuir a mesma interface da classe original, de forma que ele possa assumir o lugar dela na classe cliente de forma transparente.

Outro ponto que deve ser ressaltado é que o proxy deve possuir uma referência para um objeto da classe encapsulada, de forma a poder repassar as chamadas para ele quando necessário. Sendo assim, quando a classe cliente invocar algum método no proxy, este irá executar alguma funcionalidade e, se necessário, irá repassar a chamada para o objeto da classe encapsulada. Vale ressaltar que qualquer objeto que implementar a interface pode ser encapsulado dentro do proxy, o que permite que diversos proxies possam ser encadeados.
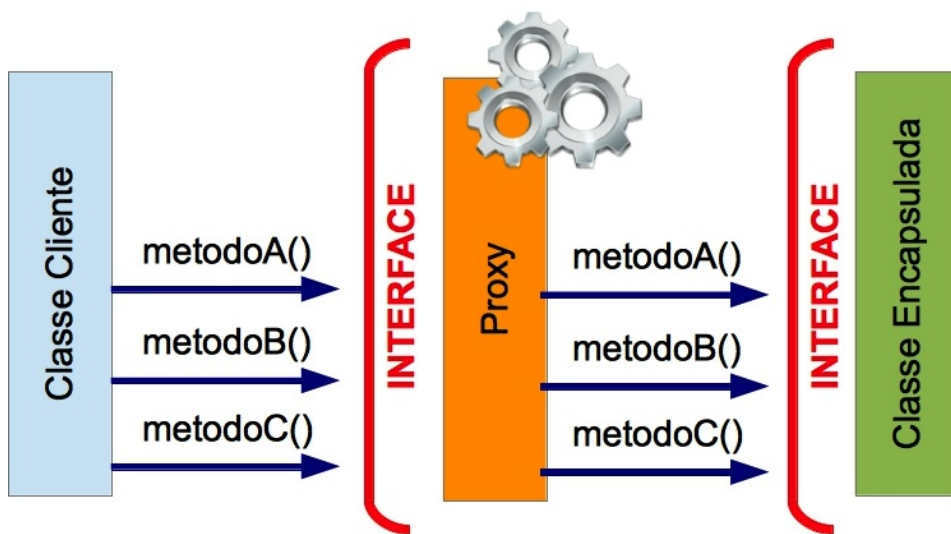
Figura 4.1: Funcionamento de um proxy

Exemplos desse tipo de proxy foram mostrados na seção 1.1, mais especificamente com as classes `LoggingAsyncProxy` e `ProdutoDAOAsync`, que imple-

mentavam proxies que executavam o método da classe encapsulada em uma thread diferente. Observe que essas classes implementam a mesma interface que o objeto que elas estão encapsulando e recebem em seu construtor. Em cada método, o proxy executa a sua funcionalidade, que no caso é criar uma nova thread, e depois invoca o método no objeto original. Para os cliente que esperam um objeto que implemente a interface, é transparente o fato do proxy estar presente ou não. Dessa forma, é como se essa funcionalidade pudesse ser plugada no objeto original.

O proxy implementado dessa forma funciona muito bem quando precisa ser aplicado a classes que compartilham uma mesma interface, pois, nesse caso, uma única implementação do proxy poderia ser utilizada para qualquer classe. O problema dessa abordagem está quando as classes que o proxy precisa encapsular não possuem uma interface em comum. Nesse caso, para cada interface diferente precisaria ser criada uma classe de proxy, o que poderia gerar repetição de código como no caso das classes `LoggingAsyncProxy` e `ProdutoDAOAsync`, que possuem um código muito similar.

4.2 Proxy dinâmico com a API Reflection

A API Reflection possui um recurso que permite a criação de um objeto que implementa uma interface em tempo de execução. Esse objeto é criado por uma classe da API, porém todas as chamadas de método recebidas por ele são redirecionadas para uma implementação da interface `InvocationHandler`. Essa implementação é quem contém a funcionalidade do proxy e redireciona as chamadas para a classe original. A seguir é mostrado passo a passo a criação de um proxy dinâmico e explicado com mais detalhe como ele funciona.

Receita para criar um proxy dinâmico

Vamos mostrar nessa seção, passo a passo como criar um proxy dinâmico com a API de reflexão. Como exemplo será mostrada a implementação de um proxy que executa os métodos da classe encapsulada de forma assíncrona, ou seja, em uma nova thread. A ideia é que essa classe implemente a mesma funcionalidade das classes `LoggingAsyncProxy` e `ProdutoDAOAsync`, porém de forma a poder ser reutilizada para qualquer interface.

Primeiro passo para implementar o proxy dinâmico é a implementação da interface `InvocationHandler`, como mostrado na listagem a seguir. Essa interface tem apenas um método chamado `invoke()`, para o qual serão direcionadas

todas as chamadas de método. Esse método recebe os três parâmetros a seguir: uma instância do objeto gerado dinamicamente (que não é o objeto encapsulado); o método que foi invocado; e um array de objetos com os parâmetros que foram passados para ele.

Listagem 4.1 - Implementação da interface InvocationHandler:

```
public class AsyncProxy implements InvocationHandler {

    public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
        return null;
    }
}
```

O segundo passo é encapsular o objeto original nessa classe. Observe na listagem a seguir, que complementa o código anterior, que foi adicionado um atributo para armazenar o objeto encapsulado, o qual é passado no construtor. Esse construtor foi definido como privado, para que ele só possa ser instanciado pele método que cria o proxy dinâmico que será definido no próximo passo. A implementação do método `invoke()` agora delega a execução para o objeto encapsulado, chamando nele o mesmo método com os mesmos argumentos.

Listagem 4.2 - Implementando o encapsulamento do objeto original:

```
public class AsyncProxy implements InvocationHandler {

    private Object obj;

    private AsyncProxy( Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
        return method.invoke(obj,args);
    }
}
```

O terceiro passo é criar um método estático que efetivamente cria o proxy dinâmico. Esse método recebe como parâmetro um objeto e retorna-o encapsu-

lado com o proxy. A criação do proxy pode ser feita a partir do método estático `newProxyInstance()` da classe `Proxy`.

Esse método recebe como primeiro parâmetro uma instância de `ClassLoader`, para a qual normalmente é passada o mesmo da classe do objeto que será encapsulado, recuperado a partir do método `getClassLoader()` de `Class`.

O segundo parâmetro são as interfaces as quais o proxy dinâmico deve ser aplicado. No exemplo, todas as interfaces do objeto são recuperadas a partir do método `getInterfaces()` e passadas como parâmetro, porém pode-se passar apenas as interfaces cujos métodos devem ser interceptados pelo proxy. O último parâmetro é uma instância de `InvocationHandler`, que no caso é uma instância da própria classe que estamos criando encapsulando o objeto passado como parâmetro.

Listagem 4.3 - Método que retorna objeto encapsulado no proxy dinâmico:

```
public class AsyncProxy implements InvocationHandler {

    public static Object criarProxy(Object obj){
        return Proxy.newProxyInstance
               (obj.getClass().getClassLoader(),
                 obj.getClass().getInterfaces(),
                 new AsyncProxy(obj));
    }

    //parte já mostrada omitida
}
```

Como último passo, falta apenas adicionar ao método `invoke()` a funcionalidade de execução assíncrona aos métodos do objeto encapsulado. Nesse exemplo, o método só será executado em uma thread diferente caso retorne `void`, porque nesse caso não é necessário aguardar sua execução para poder retornar. O código a seguir mostra a implementação dessa funcionalidade complementando o código já iniciado nas listagens anteriores.

Listagem 4.4 - Adicionando a funcionalidade de invocação assíncrona:

```
public class AsyncProxy implements InvocationHandler {

    //método de criação do proxy omitido
```

```
    private Object obj;

    private AsyncProxy( Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
        if(method.getReturnType() == void.class){
            new Thread(
                public void run(){
                    method.invoke(obj,args);
                }
            ).start();
            return null;
        } else {
            return method.invoke(obj,args);
        }
    }
}
```

É isso! São esses os passos para se criar um proxy dinâmico com a API Reflection! Os três primeiros passos são sempre muito similares e o que muda é o último passo, onde a funcionalidade adicionada pelo proxy é implementada. Essa funcionalidade, dependendo do caso, pode ser implementada antes, depois ou ao redor da invocação do método do objeto original. Observe que como temos controle dos parâmetros que são passado para o objeto original e do retorno que é dado a invocação do método, esses podem ser modificados e manipulados pelo proxy.

Repassando exceções do objeto original

Um erro comum ao se implementar um proxy dinâmico está em não se propagar corretamente as exceções lançadas pelo objeto original. Pode parecer que somente chamando o método no objeto encapsulado e passando os mesmos argumentos que o resultado seria o mesmo, porém isso não é verdade devido as exceções. É preciso lembrar que a exceção lançada pelo método original será encapsulada em uma `InvocationTargetException`.

Sendo assim, a próxima listagem mostra como seria um proxy que não faz nada, somente repassando as chamadas para o objeto encapsulado e repassando as suas respostas. Para isso, além de devolver o retorno da chamada de `invoke`, é preciso capturar a exceção `InvocationTargetException` e lançar a exceção contida

`getTargetException()`. Dessa forma, caso o objeto original lance uma exceção, o proxy irá a repassar para a classe cliente.

Listagem 4.5 - Lançando as mesmas exceções que o objeto original:

```
public class NaoFAzNadaProxy implements InvocationHandler {

    private Object obj;

    private NaoFAzNadaProxy(Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        try{
            return method.invoke(obj,args);
        } catch(InvocationTargetException e) {
            throw e.getTargetException();
        }
    }
}
```

Funcionamento de um proxy dinâmico

Para entender melhor como funciona o proxy dinâmico, a figura 4.2 apresenta uma representação de como as chamadas são realizadas. Uma confusão comum é achar que o proxy dinâmico é a implementação da interface `InvocationHandler`. Na verdade ele é uma classe criada pela máquina virtual a partir dos parâmetros passados para o método `newProxyInstance()` da classe `Proxy`. A instância do proxy será utilizada no lugar do objeto original de forma transparente para a classe cliente, visto que implementa a interface esperada por ele. Sendo assim, o proxy irá receber as chamadas de métodos normalmente como qualquer classe.
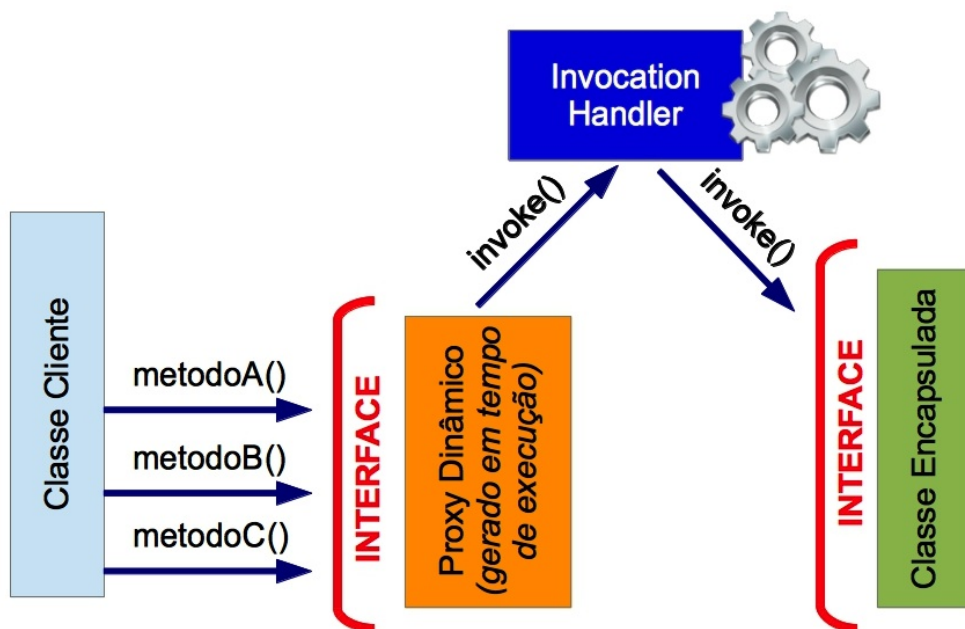
Figura 4.2: Funcionamento do proxy dinâmico

Ao receber essa chamada de método, o método `invoke()` da implementação de `InvocationHandler` é chamado. O primeiro parâmetro que é passado para esse método, como já foi dito anteriormente, é o objeto do proxy que foi gerado dinamicamente. Além disso, esse método também recebe o método e seus parâmetros, dando condições de replicar a chamada do método no objeto encapsulado. Sendo assim, a implementação de `InvocationHandler` pode executar a sua funcionalidade e repassar a chamada para o objeto encapsulado quando for adequado.

4.3 Gerando a implementação de uma interface

Um proxy dinâmico normalmente encapsula um objeto para acrescentar uma funcionalidade a ele, independente da sua interface. Porém, como pode ser visto no exemplo mostrado na seção anterior, todo esse encapsulamento é feito na implementação de `InvocationHandler`. Sendo assim, seria possível gerar um proxy dinâmico para uma interface, tratando as chamadas para todos os seus métodos, sem realmente estar encapsulando uma implementação concreta dessa interface.

Talvez você esteja se perguntando nesse momento: como vou saber qual deve ser o comportamento para os métodos de uma interface? A resposta é simples: através de seus metadados! O nome da classe, o nome do método, seus parâmetros e suas anotações podem fornecer informações suficientes para que o proxy saiba a lógica que ele precisa executar quando um determinado método for invocado. Dessa forma, a definição da assinatura do método juntamente com metadados adicionais serão responsáveis por definir seu comportamento. É como se eles definissem uma DSL interna para a definição do comportamento daquelas classes.

O que é uma DSL interna?

O acrônimo DSL significa *Domain Specific Language*, que em português seria traduzido como *linguagem específica de domínio*. Diferentemente de uma linguagem de propósito geral, como Java, que pode ser utilizada para desenvolver qualquer tipo de programa, uma DSL possui um propósito mais específico, sendo utilizada somente dentro de um domínio. Um exemplo de uma DSL é a linguagem SQL, que possui o propósito de executar operações em bases de dados relacionais.

Uma DSL interna é uma linguagem específica de um domínio que é criada dentro de uma linguagem de propósito geral já existente. No caso da criação de um proxy que gera uma implementação baseada na assinatura de um método, os elementos dessa assinatura passam a possuir uma semântica específica do domínio do proxy, que vai gerar, nesse caso, um comportamento em tempo de execução. A limitação dessa linguagem interna acaba sendo a limitação da sintaxe da própria linguagem que está sendo utilizada como base. Esse "jeito" de criar a assinatura do método, nada mais é que uma nova linguagem mais específica que está sendo criada para o domínio do proxy.

Para exemplificarmos a utilização dessa abordagem, vamos ver um exemplo de um proxy que gera a implementação de uma interface que a partir dos nomes dos métodos retorna uma propriedade de sistema. Por exemplo, se a interface possuir um método `getUserHome()`, ele irá retornar a propriedade de sistema `user.home`. Dessa forma, o proxy precisa interpretar o nome do método para extrair o nome da propriedade que precisa retornar.

O primeiro método auxiliar desenvolvido, separa a `String` com o nome do

método em um array de `String`, com os pedaços que são separados por letras maiúsculas. Por exemplo, o código transformaria `"getCamelCase"` em `{"get", "camel", "case"}`. A implementação está apresentada na listagem a seguir. Os caracteres da `String` são percorridos e adicionados na variável `corrente`, que é adicionada a lista e zerada ao se encontrar um caractere maiúsculo.

Listagem 4.6 - Método que separa a String por suas letras maiúsculas:

```
public String[] separaPorMaiusculas(String nome){
    List<String> lista = new ArrayList<>();
    String corrente = "";
    for(int i=0; i<nome.length(); i++){
        if(Character.isUpperCase(nome.charAt(i))){
            lista.add(corrente);
            corrente = "";
            corrente += Character.toLowerCase(nome.charAt(i));
        }else{
            corrente += nome.charAt(i);
        }
    }
    lista.add(corrente);
    return lista.toArray(new String[lista.size()]);
}
```

O segundo método auxiliar, apresentado a seguir, é o que recebe a `String` dividida e retorna o nome esperado para a propriedade. Ele pega cada `String` e adiciona em uma única, dividindo-as por um `"."`. Observe que o bloco `for` começa sua iteração pelo segundo elemento para pular a primeira `String` que seria o `"get"`. Por exemplo, o código transformaria `{"get", "camel", "case"}` em `"camel.case"`.

Listagem 4.7 - Transformação do array de String em nome da propriedade:

```
public String nomePropriedade(String[] strs){
    String nomeProp = "";
    for(int i=1; i<strs.length; i++){
        if(i != 1){
            nomeProp += ".";
        }
        nomeProp += strs[i];
```

```
    }
    return nomeProp;
}
```

Por fim, a classe `SystemPropertiesRetriever` na próxima listagem realiza a implementação do proxy propriamente dito. O método `invoke()` chama o método de separação por maiúsculas, extrai o nome da propriedade e a retorna, utilizando a chamada `System.getProperty()`. O método estático `criar()` que faz a criação do proxy, recebe como parâmetro o `Class` que representa a interface e retorna um objeto que a implementa dinamicamente. Observe que na chamada de `newProxyInstance()` um array com a própria interface é passado como o segundo parâmetro e uma instância da própria classe é passado como terceiro parâmetro.

Listagem 4.8 - Implementação do proxy que retorna a propriedade do sistema de acordo com o nome do método:

```
public class SystemPropertiesRetriever implements InvocationHandler {

    public static <E> E criar(Class<E> interf){
        return (E) Proxy.newProxyInstance
              (interf.getClassLoader(),
               new Class[]{interf},
               new SystemPropertiesRetriever());
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
          throws Throwable {
        String[] split = separaPorMaiusculas(method.getName());
        String nomeProp = nomePropriedade(split);
        return System.getProperty(nomeProp);
    }
}
```

Para exemplificar a utilização do proxy desenvolvido, a listagem a seguir mostra um interface que possui métodos para recuperação de diversas propriedades do sistema. A seguir, é apresentado um método `main()` que cria o proxy baseado nessa interface e imprime as propriedades do sistema no console. Vale observar que essa implementação não faz muitas verificações adicionais em relação ao método

recebido como parâmetro. O método poderia, por exemplo, possuir parâmetros ou um tipo de retorno não adequado. Deixo a implementação desse tratamento de erro como exercício aos leitores.

Listagem 4.9 - Exemplo de uso do proxy desenvolvido:

```
public interface SystemProperties {
    public String getUserCountry();
    public String getUserLanguage();
    public String getUserHome();
    public String getJavaVmSpecificationVersion();
    public String getJavaHome();
    public String getFileSeparator();
}

public class Principal {

    public static void main(String[] args) {
        SystemProperties sp =
            SystemPropertiesRetriever.criar(SystemProperties.class);
        System.out.println(sp.getUserHome());
        System.out.println(sp.getUserCountry());
        System.out.println(sp.getUserLanguage());
        System.out.println(sp.getFileSeparator());
        System.out.println(sp.getJavaHome());
        System.out.println(sp.getJavaVmSpecificationVersion());
    }
}
```

Framework Esfinge QueryBuilder

Para perceber o potencial dessa prática é interessante ver um exemplo mais avançado de utilização. O framework Esfinge QueryBuilder utiliza a assinatura dos métodos para gerar consultas para diferentes tipos de banco de dados. Para isso, são combinadas informações da assinatura do método, anotações nos parâmetros e dos próprios tipos dos parâmetros. É possível, por exemplo, definir através de uma anotação em uma interface, novos termos de domínio para serem utilizados nos nomes dos métodos. Devido a diversidade de possibilidades que se chegou com essa DSL interna, estão em desenvolvimento plugins para Eclipse que fazem a verificação do nome dos métodos em tempo de compilação e que estendem o mecanismo de refatoração de renomeação para que os nomes dos métodos sejam também modificados quando nomes nas classes persistentes forem alterados.

4.4 Proxy de classes com CGLib

"Quem não tem cão, caça com gato."
– ditado popular

Um dos problemas do proxy dinâmico da API Reflection da linguagem Java é que ele só pode ser aplicado para métodos de interfaces. Até mesmo classes que possuem uma interface, não podem ser interceptadas pelo proxy dinâmico em métodos fora dessa interface. Isso é um problema especialmente para classes nos estilo Java Beans, pois não faz muito sentido possuir uma interface para os métodos de acesso aos seus atributos. Imagine precisar criar uma interface para cada classe de domínio com todos seus métodos getters e setters somente para poder utilizar um proxy dinâmico em seus métodos.

Felizmente existe uma alternativa! A biblioteca CGLib, um acrônimo para *Code Generation Library*, possibilita a criação de proxies dinâmicos para classes a partir de geração de bytecode. Em termos de programação, o desenvolvimento de um proxy dinâmico com CGLib não é muito diferente de um proxy dinâmico com a API Reflection. Não vamos entrar muito em detalhes aqui sobre como funciona o processo de geração de bytecode e carregamento dinâmico de classes em tempo de execução, porém esse processo será melhor detalhado no capítulo **??**.

Proxy para armazenar histórico de valores

Para utilizar o CGLib em um projeto, basta adicionar o arquivo `cglib-nodep-3.1.jar` em seu classpath. Para exemplificar o uso do proxy, será desenvolvida uma classe que guarda o histórico de valores que foram atribuídos a uma propriedade de um Java Bean, permitindo sua recuperação posterior. Antes de criar o proxy, vamos definir uma interface para permitir a recuperação do histórico de valores de uma propriedade. A definição dessa interface está apresentada na listagem a seguir. Ela será utilizada para ser incorporada na classe que será gerada dinamicamente. Dessa forma, quando a chamada do método dessa interface for redirecionada para o proxy, ele retorna o valor do histórico ao invés de redirecionar a chamada.

Listagem 4.10 - Interface para recuperação do histórico:

```
public interface RecuperadorHistorico {
    public List<Object> getHistorico(String prop);
}
```

Para criar o proxy com o CGLib, a interface que precisa ser implementada é `MethodInterceptor` que possui apenas o método `intercept()`, o qual será chamado a cada chamada do método. A única diferença do método `intercept()` para o método `invoke()` de `InvocationHandler` é seu quarto parâmetro do tipo `MethodProxy`. Essa classe é uma outra forma de acessar métodos provido pelo CGLib. Por enquanto vamos ignorar esse parâmetro e utilizar a API de reflexão, mas voltaremos a isso no capítulo **??** que irá abordar outras funcionalidades do CGLib.

A próxima listagem apresenta o proxy que armazena o histórico de valores das propriedades de um Java Bean. A classe armazena como atributos o objeto encapsulado, de forma similar ao que foi mostrado para o proxy da API Reflection, e um mapa que para cada propriedade guarda uma lista de valores. Nessa versão do código, o método `intercept()` ainda está incompleta, apresentando em comentários com `PARA FAZER`, questões que serão implementadas na próxima listagem. Na implementação desse método, o primeiro condicional identifica se o método é da interface `RecuperadorHistorico`, e em caso positivo, assume-se que o método executado é o `getHistorico()`, e o histórico da propriedade passada como parâmetro deve ser retornado. No condicional seguinte, caso o método interceptado seja um setter, identificado pelo sufixo `"set"` e por receber um parâmetro, o valor do parâmetro é armazenado no histórico. Após isso, o

método do objeto encapsulado é executado normalmente.

Listagem 4.11 - Proxy para armazenamento e recuperação do histórico:

```
public class Historico implements MethodInterceptor{

    private Object encapsulated;
    private Map<String, List<Object>> historico = new HashMap<>();

    public Historico(Object encapsulated) {
        this.encapsulated = encapsulated;
    }
    public Object intercept(Object obj, Method m, Object[] args,
            MethodProxy proxy) throws Throwable {
        if(m.getDeclaringClass().equals(RecuperadorHistorico.class)){
            //PARA FAZER: retorna o histórico
        }
        if(m.getName().startsWith("set") &&
                    m.getParameterTypes().length == 1){
            String prop = deSetterParaPropriedade(m.getName());
            //PARA FAZER: armazena valor no histórico para propriedade
        }
        try {
            return m.invoke(encapsulated, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
    }
    private static String deSetterParaPropriedade(String nomeSetter){
        StringBuffer retorno = new StringBuffer();
        retorno.append(nomeSetter.substring(3, 4).toLowerCase());
        retorno.append(nomeSetter.substring(4));
        return retorno.toString();
    }
}
```

Uma questão a ser ressaltada é o tratamento especial caso o método seja da interface `RecuperadorHistorico`, que será adicionada no proxy independente da classe o implementar ou não. Esse caso especial exemplifica casos em que desejamos adicionar métodos no proxy que será gerado. Como esses método não podem ser repassados ao objeto encapsulado, eles devem ser identificados dentro do

`intercept()` e uma lógica adequada deve ser executada em resposta. Nesse caso, o resultado é um valor do proxy que deve ser retornado. Apesar desse exemplo utilizar a CGLib, uma estratégia parecida pode ser utilizada quando for necessário adicionar métodos no proxy gerado com a API Reflection.

A listagem a seguir, mostra o código do método `intercept()` completo. Dentro do primeiro condicional, que verifica se o método é da interface `RecuperadorHistorico`, é retornada a lista de valores armazenada no mapa. Observe que como nesse caso o método interceptado é conhecido, no caso `getHistorico()`, é possível saber que o primeiro parâmetro se refere a propriedade, e utilizá-lo como chave de recuperação no mapa. No segundo condicional, que deve fazer o armazenamento do histórico, é feita a verificação se já existe uma lista com os valores da propriedade. Caso não exista, uma lista é criada e inserida no mapa, e caso já exista a lista simplesmente é recuperada. Em posse da lista, o valor recebido pelo método setter como parâmetro, é inserido no histórico.

Listagem 4.12 - Completando o método intercept() do proxy Historico:

```
public Object intercept(Object obj, Method m, Object[] args,
        MethodProxy proxy) throws Throwable {
    if(m.getDeclaringClass().equals(RecuperadorHistorico.class)){
        return historico.get(args[0]);
    }
    if(m.getName().startsWith("set") &&
            m.getParameterTypes().length == 1){
        String prop = deSetterParaPropriedade(m.getName());
        List<Object> list = null;
        if(!historico.containsKey(prop)){
            list = new ArrayList<Object>();
            historico.put(prop, list);
        }else{
            list = historico.get(prop);
        }
        list.add(args[0]);
    }
    try {
        return m.invoke(encapsulated, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
```

```
}
```

A listagem a seguir mostra o método de criação do proxy, chamado de `guardar()`, onde o CGLib gera dinamicamente a classe que irá direcionar as chamadas para o `MethodInvoker`. A classe `Enhancer` gera, de certa forma, uma versão incrementada de uma certa classe. Observe que é configurado para que ele utilize a classe do objeto encapsulado como superclasse, que tenha a interface `RecuperadorHistorico` e que utilize uma instância da classe `Historico` para redirecionar as chamadas. Em seguida, é retornado um objeto dessa nova classe criada em tempo de execução. Apesar de ser diferente da geração do proxy dinâmico com a API Reflection, essa é uma receita de bolo que muda muito pouco entre diferentes proxies.

Listagem 4.13 - Método de criação do proxy com CGLib:

```
public static <E> E guardar(E obj) {
    try {
        Historico proxy =  new Historico(obj);
        Enhancer e = new Enhancer();
        e.setSuperclass(obj.getClass());
        e.setInterfaces(new Class[]{RecuperadorHistorico.class});
        e.setCallback(proxy);
        return (E) e.create();
    } catch (Throwable e) {
        throw new Error(e.getMessage());
    }
}
```

Finalizando o exemplo, a listagem a seguir mostra um exemplo de utilização do proxy criado. A classe `Produto`, um Java Bean comum e sem interfaces, será utilizada como base para a geração do proxy. Observe que para a obtenção da instância utilizada no exemplo, a chamada `Historico.guardar()` foi realizada para gerar o proxy dinâmico e encapsular o objeto passado como parâmetro. Nesse código, o valor do preço do produto é alterado diversas vezes para testarmos se o histórico está sendo realmente armazenado.

Listagem 4.14 - Exemplo de utilização do proxy:

```
public class Principal {
```

```
    public static void main(String[] args) {
        Produto p = Historico.guardar(new Produto());
        p.setNome("Design Patterns com Java");
        p.setMarca("Casa do Código");
        p.setPreco(59.90);

        //blackfriday
        p.setPreco(49.90);
        //normal
        p.setPreco(59.90);
        //natal
        p.setPreco(54.90);

        List<Object> lista =
            ((RecuperadorHistorico)p).getHistorico("preco");
        for(Object valor : lista){
            System.out.println(valor);
        }
    }
}
```

Apesar de `Produto` não implementar a interface `RecuperadorHistorico`, o proxy gerado implementa. Então, para poder acessar os métodos dessa interface, basta fazer um *cast* e invocar o método desejado, como mostrado na listagem. Executando o método `main()` é possível observar que todo histórico de valores da propriedade `preco` será impresso no console.

Esse exemplo pode parecer simples, mas com pequenas adições é possível adicionar uma referência de data ao histórico de valores, registrando o momento da modificação. A partir dessas informações, é possível gerar uma cópia do objeto com os valores das propriedades que possuía em um determinado momento. Dessa forma, é possível recuperar a versão que um determinado objeto possui a qualquer momento no tempo.

Funcionamento de um proxy gerado com CGLib

O funcionamento de um proxy dinâmico com CGLib é análogo ao proxy dinâmico criado com a API Reflection, como pode ser visto na Figura 4.3. A principal diferença é que nesse caso não existe uma interface para ser uma abstração comum entre o proxy e o objeto encapsulado. Dessa forma, a solução é que a classe gerada pelo `Enhancer` estenda a classe encapsulada, sendo, dessa forma, uma subclasse

dela. Vale ressaltar que a classe `Enhancer` permite que diversas outras questões sejam configuradas para a classe gerada, porém não está no escopo desse livro abordar essas questões.
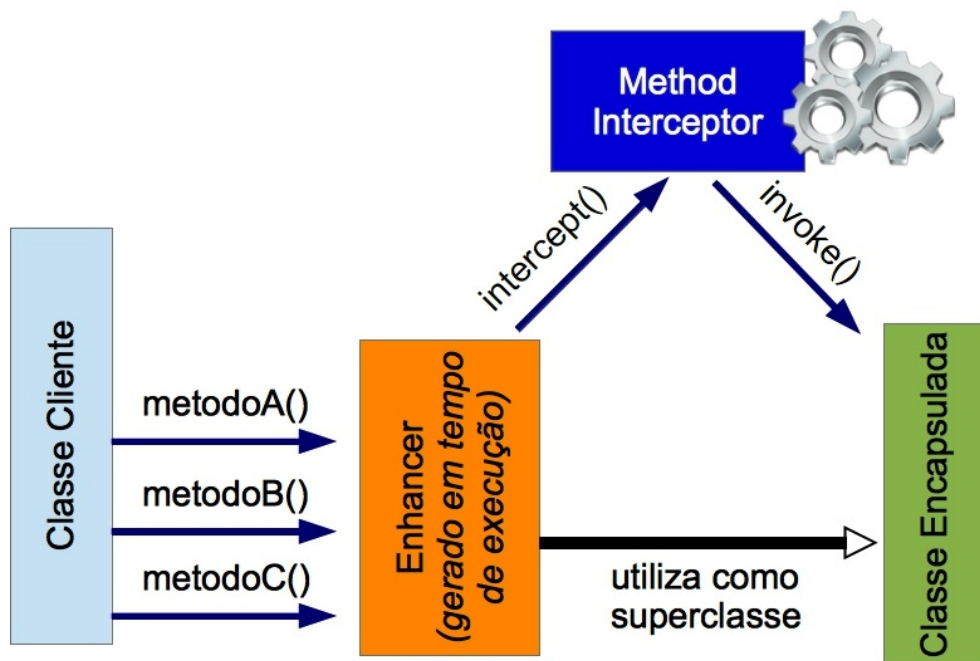
Figura 4.3: Proxy dinâmico com CGI

A cada método invocado será interceptado pela instância de `MethodInterceptor` passado no método `setCallback()` da classe `Enhancer`. De forma similar as implementações da interface `InvocationHandler`, essa classe é quem efetivamente implementa a funcionalidade do proxy dinâmico e redireciona as chamadas ao objeto encapsulado.

Maturidade do Projeto CGLib

Antes de utilizar alguma nova biblioteca em um projeto, um questionamento comum é a respeito da maturidade daquela biblioteca. Muito provavelmente se você desenvolve para aplicações corporativas já tem o CGLib no classpath da sua aplicação, pois ele é utilizado por frameworks de grande aceitação no mercado, como o Hibernate e o Spring. Sendo assim, fique tranquilo que o CGLib é uma biblioteca muito madura e já foi utilizada em frameworks que são utilizados no desenvolvimento de aplicações de grande porte.

4.5 Consumindo anotações em proxies

Nas soluções de proxy dinâmico que foram apresentadas, todas as chamadas são direcionadas para o mesmo método de tratamento. Esse método recebe o método invocado na classe original e seus parâmetros. Devido a não saber mais do método do que o que está nos seus metadados, o proxy acaba tratando todos eles da mesma forma. Nos exemplos, algumas convenções de código foram utilizadas para distinguir os métodos, porém, como foi dito na seção 3.1, a expressividade dessa abordagem é limitada.

Sendo assim, a configuração de metadados é uma excelente alternativa para distinguir os métodos de uma classe, permitindo que o proxy possa ter um comportamento distinto para cada um deles. Vamos imaginar o exemplo de um proxy que armazene o resultado de chamadas de métodos e retorne o mesmo resultado caso uma chamada semelhante seja realizada. Porém considere também que existam métodos que alteram a classe encapsulada internamente e invalidam os resultados que já foram armazenados. Ao desenvolver esse proxy, o primeiro passo seria definir anotações que configuram se deve-se fazer cache dos retornos de um método, ou se esse cache deve ser invalidado a partir da chamada daquele método. Essas anotações estão apresentadas na listagem a seguir.

Listagem 4.15 - Anotações para marcar os métodos interceptados pelo proxy:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Cache {
}
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface InvalidaCache {
}
```

Para armazenar os dados de uma invocação de métodos para o armazenamento do cache será utilizado um mapa. Esse mapa deve utilizar uma chave que identifica unicamente as chamadas de método, incluindo o método invocado e os seus parâmetros. Dessa forma, o método `gerarChave()`, apresentado na próxima listagem, cria uma `String` a partir do nome do método e do valor de seus parâmetros para serem utilizados como chave. Deixo como nota que essa geração de chaves vai depender muito dos objetos terem uma implementação adequada do método `toString()`. Para arrays, por exemplo, essa geração de chaves não funcionaria muito bem.

Listagem 4.16 - Método para geração da chave de armazenamento da invocação do método:

```
private String gerarChave(Method method, Object[] args){
    StringBuilder sb = new StringBuilder();
    sb.append(method.getName());
    for(int i=0; i<args.length; i++)
        sb.append(args[i]);
    return sb.toString();
}
```

A listagem a seguir mostra a implementação do proxy que faz o cache se baseando nas anotações dos métodos. Observe que no método `invoke()`, que nessa versão ainda está incompleto, existe um comando condicional que realiza diferentes ações dependendo da anotação presente no método. O primeiro condicional verifica a presença da anotação `@InvalidaCache`, e o cache das chamadas de método armazenadas até o momento deve ser limpo quando ele é chamado. O segundo condicional verifica a presença da anotação `@Cache`, caso no qual deve ser retornado o valor do cache, caso exista, ou executado o método no objeto encapsulado e armazenado seu resultado. Observe que se não houver nenhuma anotação presente, o método é executado normalmente.

Listagem 4.17 - Implementação do proxy que faz o cache das chamadas de método:

```
public class CacheProxy implements InvocationHandler {

    private Object obj;
    private Map<String,Object> cache = new HashMap<>();

    public CacheProxy(Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        try {
            if(method.isAnnotationPresent(InvalidaCache.class)){
                //PARA FAZER: limpar cache
            }else if(method.isAnnotationPresent(Cache.class)){
                //PARA FAZER: armazenar ou retornar objeto do cache
            }
            return method.invoke(obj, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
    }
    public static Object criar(Object obj){
        return Proxy.newProxyInstance (obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(), new CacheProxy(obj));
    }
}
```

A listagem a seguir, apresenta o método `invoke()` da classe `CacheProxy` finalizado. Observe que o método `clear()` do mapa que armazena o cache é chamado para limpar o cache na presença da anotação `@InvalidaCache`. Já na presença da anotação `@Cache`, o primeiro passo é gerar a chave relativa a invocação utilizando o método `gerarChave()` mostrado anteriormente. Em seguida, é verificado se o mapa que armazena o cache já possui a chave. Caso a chave exista, o valor armazenado no mapa é retornado e o método no objeto encapsulado nem é invocado. Por outro lado, se a chave não existir, o objeto encapsulado é invocado, o seu retorno e armazenado no mapa e, por fim, esse valor é retornado pelo proxy.

Listagem 4.18 - Implementação do método invoke() do CacheProxy:

```
public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
```

```
    try {
        if(method.isAnnotationPresent(InvalidaCache.class)){
            cache.clear();
        }else if(method.isAnnotationPresent(Cache.class)){
            String chave = gerarChave(method, args);
            if(cache.containsKey(chave)){
                return cache.get(chave);
            }else{
                Object retorno = method.invoke(obj, args);
                cache.put(chave, retorno);
                return retorno;
            }
        }
        return method.invoke(obj, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
}
```

Para finalizar o exemplo, vamos mostrar como uma interface seria anotada para a utilização do proxy. A listagem a seguir mostra a interface `Operacao` que possui as anotações em seus métodos. A ideia é que essa interface seja implementada por classes que armazenam um valor internamente e executem operações nesse valor retornando o resultado. Nesse caso, métodos `somar()` e `multiplicar()` que retornam um valor baseado no valor interno e no parâmetro podem ter seus retornos armazenados no cache. Por outro lado, o método `mudar()` que altera o valor interno deve invalidar o cache, pois os retornos armazenados internamente deixam de ser válidos.

Listagem 4.19 - Anotações para marcar os métodos interceptados pelo proxy:

```
public interface Operacao<E> {
    @InvalidaCache public void mudar(E obj);
    @Cache public E somar(E obj);
    @Cache public E multiplicar(E obj);
}
```

Esse proxy desenvolvido nesse exemplo pode ser muito útil para classes que realizam chamadas remotas ou executa operações em uma base de dados. Nesse caso, chamadas que recuperam dados baseadas no estado do banco ou do objeto remoto

podem ser adicionadas no cache e as que alteram esse estados, devem ser utilizadas para invalidar o cache. De qualquer forma, antes de decidir utilizar um cache desse tipo em uma aplicação real é importante fazer uma medição de desempenho baseada em cenários de uso do software, pois apesar do cache poder acelerar o tempo médio de retorno de um método, os resultados armazenados ocupam memória. Sendo assim, para valer a pena, o ganho em velocidade deve compensar esse espaço utilizado.

E se eu quiser as anotações do método da classe e não da interface?

Uma observação importante em relação ao proxy dinâmico da API Reflection é que a instância de `Method` passada como parâmetro é da interface que está sendo encapsulada, e não do objeto que foi encapsulado. Como eles possuem a mesma assinatura, muitas vezes isso não tem muita importância, porém essa questão passa a ser relevante quando estamos buscando anotações, pois nesse caso importa se foram adicionadas na interface ou na classe. Porém, o fato do método passado ser o da interface, não significa que devemos sempre utilizar anotações na interface. Quando os metadados forem referentes a abstração devem ser adicionados na interface, porém quando forem específicos da implementação devem ser adicionados na classe.

Com base no método recebido como parâmetro, não é difícil de recuperar o mesmo método declarado na classe. Imaginando que a variável `obj` represente o objeto encapsulado e a variável `m` represente o método recebido como parâmetro, o método da classe poderia ser recuperado com a chamada `obj.getClass().getMethod(m.getName(),m.getParameterTypes())`. Assim as anotações poderiam ser facilmente recuperadas da declaração do método na classe e utilizadas no proxy.

4.6 Outras formas de interceptar métodos

"As laranjas não são as únicas frutas"
– título de um livro de Jeanette Winterson

Apesar de estar fora do escopo desse livro, acho importante dizer que existem

outras formas de interceptar a execução de métodos em Java. Dependendo da plataforma para a qual você está desenvolvendo ou do tipo de componente que está sendo criado, a própria API pode fornecer uma alternativa para interceptação dos métodos. Nesse caso, os mesmos conceitos apresentados para a criação dos proxies dinâmicos apresentados aqui podem ser utilizados. O objetivo dessa seção é apresentar alguns exemplos dessas outras tecnologias para

Por exemplo, na plataforma Java EE, existe um tipo de componente chamado *interceptor* que funciona de forma similar a um proxy dinâmico, porém para instâncias que são gerenciadas pelo container Java EE. Para ser um interceptor, a classe precisa receber a anotação `@Interceptor` e possuir um método anotado com `@AroundInvoke`, que necessariamente precisa receber um parâmetro do tipo `InvocationContext` e retornar `Object`. A instância de `InvocationContext` é que contém todas as informações relacionadas a invocação, como o método interceptado e os parâmetros recebidos.

Listagem 4.20 - Exemplo de Interceptor da plataforma Java EE:

```
@Interceptor
public class ExemploInterceptor {

       @AroundInvoke
       public Object interceptaMetodo(InvocationContext ctx)
               throws Exception {
           //implementa lógica a ser acrescentada
       }
}
```

Como os objetos do container Java EE são criados por ele, não é necessário implementar nenhum método que insere o *interceptor* na classe. Uma forma de ligar um interceptor em uma classe é através da anotação `@Interceptors`, que recebe as classes de interceptors que devem atuar em uma class ou em um método. Uma outra forma mais sofisticada é através da criação de anotações que fazem a ligação entre os interceptors e as classes interceptadas. Para criar essa anotação de ligação, ela precisa ser anotada com `@InterceptorBinding` e deve ser adicionada na própria classe do interceptor. Sendo assim, as classes e métodos que receberem essa anotação serão interceptadas.

Uma outra forma de interceptação de métodos é utilizando a programação orientada a aspectos. A orientação a aspectos possui outras coisas além da inter-

ceptação de métodos, porém aqui iremos focar mais exclusivamente nessa parte. A principal implementação da orientação a aspectos em Java é o **AspectJ**, o qual disponibiliza duas diferentes opções de sintaxe, sendo uma delas uma sintaxe própria e a outra baseada em anotações. A listagem a seguir exemplifica a sintaxe baseada em anotações.

Listagem 4.21 - Exemplo de aspecto com AspectJ utilizando a sintaxe de anotações:

```
@Aspect
public class AspectoRegistro{

    @Before("execution(* br.casadocodigo.*.*(..))")
    public void antes(JoinPoint jp) {
        //executa antes
    }
    @After("execution(* br.casadocodigo.*.*(..))")
    public void depois(JoinPoint jp) {
        //executa depois
    }
    @Around("execution(* br.casadocodigo.*.*(..))")
    public Object emVolta(ProceedingJoinPoint jp) throws Throwable{
        //executa antes
        Object returno = jp.proceed();
        //executa depois
        return returno;
    }
}
```

As anotações `@Before`, `@After` e `Around` configuram o método do aspecto, que nesse caso é chamado de adendo ou *advice*, para executar respectivamente antes, depois ou a volta do método interceptado. Cada anotação dessa recebe como parâmetro uma expressão que define um conjunto de junção, ou *pointcut*, que nada mais é que o conjunto de métodos que serão interceptados por aquele adendo do aspecto. O grande diferencial dos aspectos em relação as outras técnicas é que os métodos interceptados são definidos no aspecto, havendo um desconhecimento da aplicação da sua existência. A ligação das classes com os aspectos pode ocorrer na compilação, quando as classes são carregadas, ou mesmo em tempo de execução com o uso do framework Spring. O capítulo **??** irá apresentar uma outra funcionalidade do AspectJ que pode ser utilizada para o mapeamento de anotações.

Quero mais uma vez deixar claro que essa seção foi mesmo para apresentar a existência de outras abordagens, que acabam sendo um pouco mais específicas. Não é o objetivo ensinar a utilização dessas técnicas como no restante do capítulo. Acredito que entendendo melhor essas outras implementações é possível ter um panorama geral de como esse tipo de funcionalidade é implementado em diferentes contextos. Assim, se você estiver trabalhando com alguma plataforma ou framework que dê suporte a esse tipo de funcionalidade, vai saber mais facilmente os conceitos vistos nesse capítulo podem ser encaixados lá.

4.7 Considerações finais

Os proxies dinâmicos são um recurso poderoso da reflexão que podem ser utilizados para adicionar comportamento em classes existentes, e ainda fazer isso de forma reutilizável para qualquer interface. Como foi visto, para implementar um proxy dinâmico não é muito complicado, bastando seguir uma receita. Esse capítulo procurou explorar diferentes tipos de proxy dinâmico, desde o que não encapsula nenhum objeto até o que faz uso de metadados para diferenciar os métodos interceptados. No final do capítulo mostrou-se rapidamente algumas outras abordagens para a interceptação de métodos.

Esse capítulo finaliza a primeira parte do livro onde os conceitos básicos de reflexão e do uso de metadados são apresentados. Nesses primeiros capítulos também foram apresentadas as APIs básicas que são utilizadas em Java para a utilização desses conceitos. Chegando nesse ponto, espero que os leitores já estejam familiares ao uso da reflexão e já consigam dar suas primeiras pinceladas em componentes que a utilizam. Prepare-se que na próxima parte desse livro você vai aprender o que se precisa para fazer uma obra prima...

Parte II

Boas Práticas

Nos capítulos anteriores do livro foram abordadas as APIs e recursos da linguagem que podem ser empregados para o uso de reflexão da linguagem Java. Porém, se sua intenção é realmente criar um componente ou framework reutilizável com reflexão e metadados, conhecer as classes e métodos que serão utilizados é só o começo. Faço um paralelo de quando se aprende herança e polimorfismo em Java, pois saber somente isso não é suficiente para criar um bom design orientado a objetos.

Essa parte do livro apresenta boas práticas que podem ser utilizadas no desenvolvimento desse tipo de componente. Os próximos capítulos, compilam de uma forma fluida e didática, algumas práticas descobertas como parte da pesquisa que realizei e venho realizando. Essa pesquisa envolve não somente a análise das práticas utilizadas em diversos frameworks existentes, como também sua utilização no desenvolvimento de novos frameworks para confirmar a adequabilidade da aplicação de cada uma delas.

Capítulo 5

Testando classes que usam Reflexão

"Um desenvolvedor como eu se importa com escrever código novo e fazê-lo o mais interessante e eficiente possível. Mas muito poucas pessoas querem fazer os testes."
– Linus Torvalds

Como foi dito anteriormente, existem vários erros que são pegos já em tempo de compilação que quando lidamos com reflexão podem acontecer em tempo de execução. Sendo assim, pelos próprios exemplos que foram mostrados nos capítulos anteriores, é possível perceber que se a classe ou método não possui a estrutura esperada pelo componente, muitas coisas podem dar errado. Por exemplo, pode-se tentar instanciar uma classe com um construtor que ela não possui ou invocar um método com a quantidade errada de parâmetro. As classes desenvolvidas com reflexão devem estar preparadas para lidar com qualquer tipo de classe, e existem muitas coisas possíveis de acontecer. Uma premissa errada ou alguma estrutura inesperada, pode causar um bug sério na aplicação que utiliza o componente.

O fato de um componente ou classe que utilize reflexão poder ser reutilizado em diferentes contextos dentro de uma aplicação, apesar de ser uma coisa muito boa,

aumenta a preocupação com a qualidade desse código. Muitas vezes, mesmo que a classe já tenha sido reutilizada várias vezes, uma situação distinta ainda pode causar erros. Além disso, é preciso não somente que o componente que utiliza reflexão esteja correto, como também que a classe da aplicação tenha a estrutura adequada e possua os metadados configurados de forma a gerar o comportamento desejado.

Dessa forma, esse capítulo se dedica a abordar questões relativas ao teste de classes e componentes que utilizam reflexão, abordando também o teste das aplicações que utilizam esses componentes. Vou considerar que os leitores estejam familiarizados com os conceitos de testes automatizados e como criá-los com o JUnit 4 (para os que não conhecem, leiam o quadro). Porém, se você está lendo um livro avançado de programação como esse e não sabe criar testes de unidade ainda, acho que você está em apuros! Sugiro fortemente que corra atrás desse conhecimento, porém acredito que com um conhecimento básico de JUnit é possível acompanhar o livro.

JUnit 4 em menos de um minuto!

Para criar testes de unidade com o JUnit 4, o primeiro passo é adicionar sua biblioteca no classpath. Para implementar os testes, crie uma classe comum e adicione métodos que realizam os testes. Para ser considerado um teste, o método precisa ser anotado com `@Test`. Cada teste deve preparar o cenário do teste, executar a funcionalidade que deseja-se testar o comportamento e verificar se o comportamento foi o esperado. Para verificar o comportamento, a classe `Assert` possui diversos métodos estáticos que permitem fazer comparações entre valores. A listagem a seguir mostra um exemplo simples de classe de teste.

Listagem 5.1 - Exemplo simples de funcionamento do JUnit:

```
public class ClassDeTeste {
    @Test
    public void metodoDeTeste() {
        ClassTestada testada = new ClassTestada();
        String resultado = testada.metodoTestado();
        assertEquals("valor esperado",resultado);
    }
}
```

O JUnit possui um grande suporte de todos os IDEs existentes, onde para rodar uma classe de testes, basta clicar nela e solicitar sua execução. Isso é o suficiente para acompanhar os exemplos! Caso algo diferente seja utilizado, a própria seção irá explicar o funcionamento.

Para mostrar como fazer os testes automatizados desse tipo de classe, esse capítulo vai focar muito em exemplos. No caso, vamos pegar classes desenvolvidas nos capítulos anteriores e mostrar como poderia ser feito o teste para elas. Os exemplos escolhidos possuem características distintas, justamente para mostrar a diversidade de situações que se pode encontrar ao se testar esse tipo de componente de software.

5.1 Variando estrutura da classe para teste

Quando temos uma classe ou componente baseado em reflexão e metadados, normalmente o seu comportamento vai depender da classe passada como parâmetro. Sendo assim, para testá-los é preciso passar classes com diferentes estruturas como parâmetro. Dessa forma, o cenário de cada teste vai envolver a criação de uma nova classe, e quando cabível vai criar um objeto dessa classe. A ação que define a funcionalidade que será definida nesse teste envolve a invocação do componente que utiliza reflexão. Por fim, a verificação irá checar se o comportamento foi o esperado, o que pode envolver a verificação do valor de um retorno, do estado de um objeto, ou mesmo se um determinado método foi invocado.

Para começar, vamos implementar os testes automatizados do gerador de mapas apresentado na seção 1.3. Para esse componente, o teste mais simples seria gerar o mapa de propriedades de um Java Bean simples, com algumas propriedades e sem anotações. A próxima listagem mostra como esse primeiro teste seria implementado. Inicialmente, definimos a classe **Bean** que será consumida pela classe de geração de mapas. Observe que ela possui dois atributos com métodos de acesso getters e setters. Em seguida é criada uma instância dessa classe, as suas propriedades são configuradas e, por fim, o objeto é passado ao método `gerarMapa()`. Para verificar se o comportamento foi correto, são inseridas suas asserções que verificam se o mapa retornado possui as propriedades esperadas.

Listagem 5.2 - Teste da funcionalidade básica do gerador de mapas:

```
public class TesteGeradorMapa {
    @Test
    public void mapaDeClasseSimples() {

        class Bean{
            private String prop1;
            private int prop2;
            public String getProp1() {
                return prop1;
            }
            public void setProp1(String prop1) {
                this.prop1 = prop1;
            }
            public int getProp2() {
                return prop2;
```

```
            }
            public void setProp2(int prop2) {
                this.prop2 = prop2;
            }
        }

        Bean b = new Bean();
        b.setProp1("teste");
        b.setProp2(25);
        Map<String,Object> mapa = GeradorMapa.gerarMapa(b);
        assertEquals("teste",mapa.get("prop1"));
        assertEquals(25,mapa.get("prop2"));
    }
}
```

Como foi descrito, é preciso definir uma classe para a realização da verificação. No exemplo mostrado, ela é criada dentro do próprio método de teste, sendo válida sua utilização somente dentro de seu escopo. As principais vantagens dessa abordagem é que a classe definida fica próxima ao seu uso, e que o mesmo nome pode ser utilizado em outros métodos sem haver conflito. Porém, essa é uma das possibilidades, e ela pode ser definida em outros escopos, como fora do método ou em seu próprio arquivo. Uma classe definida em um contexto mais amplo pode ser utilizada em mais métodos de testes, mas por outro lado, a definição dentro do mesmo arquivo aumenta a legibilidade do código tornando mais fácil de ver o cenário que está sendo definido. Sendo assim, é preciso pesar essas questões para ver qual o local mais adequado para sua definição.

Continuando a definição dos testes, a próxima listagem verifica a utilização da anotação `@Ignorar`. A diferença na definição da classe `Bean` para esse teste está na adição da anotação `@Ignorar` no método `getProp1()`. Observe que a criação do objeto e a definição da propriedade são iguais às do método anterior. Nas asserções, é verificado se o mapa não possui a chave `"prop1"` através do método `assertFalse()`, e a outra verificação é a mesma que já havíamos feito.

Listagem 5.3 - Testando o uso de anotação @Ignorar:

```
@Test
public void ignoraPropriedade() {

    class Bean{
```

```
        private String prop1;
        private int prop2;
        @Ignorar
        public String getProp1() {
            return prop1;
        }
        public void setProp1(String prop1) {
            this.prop1 = prop1;
        }
        public int getProp2() {
            return prop2;
        }
        public void setProp2(int prop2) {
            this.prop2 = prop2;
        }
    }

    Bean b = new Bean();
    b.setProp1("teste");
    b.setProp2(25);
    Map<String,Object> mapa = GeradorMapa.gerarMapa(b);
    assertFalse(mapa.containsKey("prop1"));
    assertEquals(25,mapa.get("prop2"));
}
```

Por fim, o teste seguinte apresentado na próxima listagem exercita o caso em que a anotação `@NomePropriedade` é utilizada para mudar o nome da chave usada no mapa. Observe que, nesse caso, as asserções verificam se existe a entrada no mapa com o nome configurado na anotação e se não existe a chave com o nome do atributo.

Listagem 5.4 - Testando a mudança de nome da propriedade do mapa:

```
@Test
public void mudaNomePropriedade() {

    class Bean{
        private String prop1;
        private int prop2;
        public String getProp1() {
            return prop1;
```

```
        }
        public void setProp1(String prop1) {
            this.prop1 = prop1;
        }
        @NomePropriedade("propriedade2")
        public int getProp2() {
            return prop2;
        }
        public void setProp2(int prop2) {
            this.prop2 = prop2;
        }
    }

    Bean b = new Bean();
    b.setProp1("teste");
    b.setProp2(25);
    Map<String,Object> mapa = GeradorMapa.gerarMapa(b);
    assertEquals("teste",mapa.get("prop1"));
    assertEquals(25,mapa.get("propriedade2"));
    assertFalse(mapa.containsKey("prop2"));
}
```

Esse exemplo demonstra o processo básico de se testar uma classe que utiliza reflexão. Como foi mostrado no exemplo desses três testes, a criação do objeto e a invocação dos métodos acabam sendo as mesmas, e o que muda no processo de teste é a estrutura da classe passada como parâmetro. A ideia é utilizar uma classe como base e ir modificando a estrutura de forma a mudar o comportamento esperado.

Verificando chamadas de método

Um tipo de funcionalidade frequentemente feita por classes que utilizam reflexão envolve a invocação de métodos em classes que ela só reconhece em tempo de execução. Sendo assim, um teste que precisará ser feito para validar esse tipo de funcionalidade envolve, como parte da verificação, assegurar que os métodos corretos foram invocados. Isso implica não somente verificar a invocação dos métodos certos, mas também se certificar que outros métodos não foram invocados.

Aqui, utilizaremos como exemplo o validador de objetos apresentado da seção 2.4. Uma das funcionalidades dessa classe envolve a invocação de métodos começados com `"validar"` e que não recebam parâmetros. Sendo assim, a próxima listagem mostra dois testes para verificar essa funcionalidade. Observe

que o primeiro teste verifica se o método correto é invocado, e o segundo verifica se somente métodos que obedecem aos requisitos são invocados. Para verificar se a invocação é feita ou não, é criada uma variável booleana na classe para teste, que é alterada no momento que o método é invocado. Sendo assim, vendo o valor dessa variável conseguimos ver se o método foi invocado ou não.

Listagem 5.5 - Testando a invocação de métodos no validador:

```
public class TesteValidador {

    @Test
    public void testInvocaMetodo() throws ValidacaoException {
        class ParaValidar{
            public boolean invocou = false;
            public void validarInformacao(){
                invocou = true;
            }
        }
        ValidadorObjetos v = new ValidadorObjetos();
        ParaValidar pv = new ParaValidar();
        v.validarObjeto(pv);
        assertTrue(pv.invocou);
    }
    @Test
    public void naoInvocaMetodo() throws ValidacaoException {
        class ParaValidar{
            public boolean invocouA = false;
            public boolean invocouB = false;
            public void validarParamErrado(String s){
                invocouA = true;
            }
            public void nomeErrado(String s){
                invocouB = true;
            }
        }
        ValidadorObjetos v = new ValidadorObjetos();
        ParaValidar pv = new ParaValidar();
        v.validarObjeto(pv);
        assertFalse(pv.invocouA);
        assertFalse(pv.invocouB);
    }
```

```
}
```

Uma outra funcionalidade da classe `ValidadorObjetos` envolve a invocação de atributos cujo tipo implemente a interface `Validador`. Nesse caso, a ideia é a mesma do teste mostrado anteriormente, porém como a invocação é feita no objeto que está no atributo, o mecanismo deve ser adicionado nesse objeto, e não na classe que está sendo criada. A listagem a seguir mostra a implementação do teste dessa funcionalidade.

Observe que a classe `ValidadorTeste` é criada para simular uma classe que implementa a interface `Validador` e possui um atributo booleano para verificar se o método foi invocado ou não. Em seguida, dentro do método de teste, é criada uma classe com três atributos, dos tipos `ValidadorTeste`, `Validador` e `Object`, sendo que todos recebem uma instância de `ValidadorTeste`. Segundo os requisitos, para ser invocado, o tipo do atributo deve implementar a interface `Validador`, o que é refletido pelas asserções, onde somente o atributo do tipo `Object` não deve ser invocado.

Listagem 5.6 - Teste da invocação de atributos do tipo Validador:

```
//Declaração fora da classe
public class ValidadorTeste implements Validador {
    public boolean invocou;
    public void validar(Object o) throws Exception {
        invocou = true;
    }
}

//método de teste
@Test
public void invocaAtributos() throws ValidacaoException {
    class ParaValidar{
        public ValidadorTeste v1 = new ValidadorTeste();
        public Validador v2 = new ValidadorTeste();
        public Object v3 = new ValidadorTeste();
    }
    ValidadorObjetos v = new ValidadorObjetos();
    ParaValidar pv = new ParaValidar();
    v.validarObjeto(pv);
    assertTrue(pv.v1.invocou);
    assertTrue(((ValidadorTeste)pv.v2).invocou);
```

```
    assertFalse(((ValidadorTeste)pv.v3).invocou);
}
```

Posso utilizar mock objects nesse tipo de teste?

Se você é um desenvolvedor que é familiar com a criação de testes, talvez deva estar se perguntando se essa criação de classes mostrada nas listagens anteriores não seria o mesmo que criar um *mock object*. Um mock object é um objeto que é criado para simular uma dependência da que está sendo classe testada. Essa simulação envolve dar as respostas para simular diferentes cenários de teste e verificar se as invocações de método foram feitas de acordo com as esperadas. Em uma classe que não utiliza reflexão, a dependência possui um contrato esperado, determinado por uma classe ou uma interface. Sendo assim, o mock object implementa esse mesmo contrato de forma a poder substituir essa dependência.

No caso do teste de classes que utilizam reflexão, o objeto utilizado como dependência não possui uma interface definida. Sendo assim, nesse caso o que precisa ser feito em cada caso de teste é a definição de classes com estruturas diferentes, e não comportamentos diferentes para a mesma estrutura. Apesar de isso não ser um termo tão usual, um artigo científico definiu esse tipo de classe criada para teste como *mock class* [1]. Sendo assim, o mock object se refere à simulação de comportamento para a dependência da classe testada e mock class, à simulação de estruturas diferentes de classes para teste. De certa forma, é possível criar um mock object de um mock class, pois diferentes testes podem usar uma mesma estrutura de classe mas com diferentes comportamentos. Isso é comum quando a mock class representa uma interface e o mock object é uma implementação dela.

Nos exemplos mostrados, cada teste criava sua mock class já com o comportamento esperado para o teste. Muitas vezes, como uma nova classe precisará ser definida de qualquer forma, é mais fácil já defini-la fazendo o necessário para gerar o cenário do teste. O melhor exemplo de mock object nos testes já apresentados nesse capítulo foi a classe `ValidadorTeste`, pois, nesse caso, a interface esperada é fixa, por mais que os atributos sejam lidos por reflexão.

No caso dos mock objects, uma alternativa à implementação manual de novas classes para a simulação de comportamento são os frameworks de criação de mocks. Esses frameworks normalmente utilizam proxies dinâmicos para gerar o comportamento esperado pela classe e verificar se as chamadas realizadas foram as esperadas. Exemplos de frameworks Java que implementam esse tipo de funcionalidade são o **JMock**, o **EasyMock** e o **Mockito**. A vantagem em utilizar esse tipo de framework

está em não necessitar criar diversas classes para simular diferentes comportamentos para os testes, permitindo que a definição do comportamento e verificações do mock sejam feitos no próprio método de teste.

Os exemplos desse livro irão utilizar o JMock para mostrar a utilização desse tipo de framework em testes de classe que utilizam reflexão, porém está fora do escopo desse livro explicar o funcionamento desse framework.

JMock em menos de cinco minutos!

Para utilizar o JMock, a classe de teste deve possuir algum atributo que tenha um tipo derivado da classe `Mockery`, que será utilizado na criação dos mocks. Em versões mais antigas do JUnit, a classe de teste era anotada com `@RunWith(JMock.class)` para que as verificações dos mocks rodassem depois do método de teste. Em versões mais novas basta criar um atributo do tipo `JUnitRuleMockery` e o anotar com `@Rule`. Para criar o mock de uma interface basta utilizar o método `mock()` passando o `Class` correspondente, sendo que se for criar mais de um mock da mesma classe precisa passar um parâmetro adicional com uma `String` que nomeia a instância do mock.

Para definir as expectativas e comportamento do mock, a chamada `checking(new Expectations(){{ ... }})` é utilizada, sendo que todas as definições são feitas no lugar dos três pontos. O JMock utiliza uma DSL interna para definir as chamadas esperadas e seu comportamento, incluindo o número de invocações, ordem de invocação, parâmetros esperados e o resultado da invocação, como retorno ou lançamento de exceção. As expectativas utilizadas em cada exemplo do livro serão explicadas com detalhe, possibilitando que mesmo quem não conheça o framework possa acompanhá-los. No site do projeto JMock, existe um resumo de sua sintaxe em uma *cheat sheet* no endereço http://jmock.org/cheat-sheet.html. A listagem a seguir mostra a estrutura básica de uso do JMock.

Listagem 5.7 - Exemplo de uso do JMock:

```
public class TesteComJMock {

    @Rule
    public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
    public void testeComMock() throws Exception {
        ClasseMock mock= ctx.mock(ClasseMock.class);
        ClasseTestada c = new ClasseTestada();
        c.setClasseMock(mock);

        ctx.checking(new Expectations() {{
            //expectativas do mock
            //retornos dados pelo mock
        }});
```

Para vermos como o teste usando mock object pode ser feito simulando os objetos da interface `Validador`, a próxima listagem mostra como o framework JMock poderia ser utilizado. Para que os mocks possam ser criados e suas expectativas verificas no final de cada teste, a classe de teste deve ter um atributo do tipo `JUnitRuleMockery` anotada com `@Rule`. No método de teste, observe que a mock class `ParaValidar` é criada recebendo mock objects em seus dois atributos, a partir da chamada do método `mock()` da class do JMock. Em seguida, são definidas as expectativas: o método `validar()` do atributo de tipo `Validador` deve ser invocado uma vez e o método `validar()` do atributo de tipo `Object` não deve ser invocado.

Listagem 5.8 - Uso de framework de mock para criação de teste com Validador:

```
public class TesteValidador {

    @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
    public void invocaAtributosJMock() throws Exception {

        class ParaValidar{
            public Validador v1 = ctx.mock(Validador.class, "v1");
            public Object v2 = ctx.mock(Validador.class, "v2");
        }

        ValidadorObjetos v = new ValidadorObjetos();
        final ParaValidar pv = new ParaValidar();

        ctx.checking(new Expectations() {{
            oneOf (pv.v1).validar(pv);
            never (((Validador)pv.v2)).validar(pv);
        }});

        v.validarObjeto(pv);
    }
}
```

Lidando com testes que envolvem exceções

As exceções são um tipo de retorno que uma invocação de método pode dar como resultado, significando a existência de algum tipo de erro na execução da funcionalidade. A exceção lançada pode indicar uma situação em que a funcionalidade não pode ser executada, mas que faz parte de um cenário comum de negócio. Exemplos são informações inválidas fornecidas pelo usuário, uma autenticação falha ou a tentativa de acesso a uma funcionalidade à qual o usuário não é autorizado. A ocorrência dessas exceções não significa que existe um erro no software, mas que uma execução da funcionalidade não pode ser completada devido a uma situação excepcional prevista.

Outro tipo de exceção é aquele que indica que o software não está funcionando corretamente devido a um problema interno, que representa um erro ou um bug. Muitas classes e componentes possuem uma abordagem defensiva, na qual não confiam em outras classes e verificam se as chamadas estão sendo feitas da forma correta. Nesse caso, o lançamento de uma exceção significa que o uso daquela classe não está sendo feito da forma correta. Nesse caso, é importante que a mensagem de erro seja bem descritiva para possibilitar que o desenvolvedor detecte a falha e possa corrigi-la. Em caso de classes que utilizam reflexão, esse tipo de verificação é muito importante pois a classe recebida como parâmetro pode possuir uma estrutura inadequada ou erros na configuração de metadados.

Para exemplificar o primeiro tipo de exceção, serão utilizados os próprios testes da classe `ValidadorObjetos()`. Como foi mostrado em seu desenvolvimento, essa classe deve pegar as exceções lançadas pelos métodos e classes de validação, encapsulando-as em uma lista que é lançada em uma `ValidaçãoException`. A listagem a seguir mostra um teste que simula um cenário em que uma exceção é lançada de um método de validação, chamado `validarInformacao()`, e uma implementação de `Validator` da qual é criado um mock com o framework JMock. Observe que para fazer o mock lançar a exceção, nas expectativas é utilizada a chamada `will(throwException(...))`.

Listagem 5.9 - Verificando informações das exceções recebidas:

```
@Test
public void capturaExcecoes() throws Exception {

    class ParaValidar{
        public Validador v = ctx.mock(Validador.class);
```

```
        public void validarInformacao() throws Exception{
            throw new Exception("mensagem A");
        }
    }

    ValidadorObjetos v = new ValidadorObjetos();
    final ParaValidar pv = new ParaValidar();

    ctx.checking(new Expectations() {{
        oneOf (pv.v).validar(pv);
        will(throwException(new Exception("mensagem B")));
    }});

    try {
        v.validarObjeto(pv);
        fail();
    } catch (ValidacaoException e) {
        List<Exception> erros = e.getErros();
        assertEquals("mensagem A", erros.get(0).getMessage());
        assertEquals("mensagem B", erros.get(1).getMessage());
    }
}
```

Nesse caso, como existem verificações para serem feitas na exceção lançada, o método `validarObjeto()` precisa ser envolto em um bloco `try`. Como é esperado que uma exceção seja lançada, o método `fail()` é invocado logo após a chamada de `validarObjeto()`, pois se ela não for lançada o teste deve falhar. Dentro do bloco `catch`, a lista de exceções é recuperada e é verificado, através da mensagem de erro, se as exceções lançadas pelo método da classe e pelo mock estão na lista.

Como exemplo de teste de exceções que são lançadas por uma estrutura inadequada da classe, serão utilizados testes da `MapeamentoParametros` apresentada na seção 3.5. Esse exemplo foi escolhido porque foram feitas várias verificações no formato da classe, lançando exceções quando a sua estrutura não condizia com o que era esperado. Na listagem a seguir mostra o exemplo de dois testes que fazem essa verificação. O primeiro verifica o caso em que o tipo da propriedade seja inteiro, sendo que o esperado é somente `String`, e o segundo explora o caso de o método setter receber dois parâmetros.

Listagem 5.10 - Exemplo de testes que verificam classes com estrutura inadequada:

```
public class TesteMapeamentoParametros {

    @Test(expected=MapeamentoException.class)
    public void testTipoErrado() {

        class Param{
            @Parametro("-i")
            private int info;
            public int getInfo() {
                return info;
            }
            public void setInfo(int info) {
                this.info = info;
            }
        }

        String[] a = {"-i","23"};
        MapeamentoParametros<Param> map =
            new MapeamentoParametros<>(Param.class);
        Param p = map.mapear(a);
    }

    @Test(expected=MapeamentoException.class)
    public void testQuantidadeParamErrado() {

        class Param{
            @Parametro("-i")
            private String[] info;
            public String[] getInfo() {
                return info;
            }
            public void setInfo(String infoA, String infoB) {
                this.info = new String[]{infoA, infoB};
            }
        }

        String[] a = {"-i","textoA","textoB"};
        MapeamentoParametros<Param> map =
```

```
            new MapeamentoParametros<>(Param.class);
        Param p = map.mapear(a);
    }
}
```

Nesse caso, não existe nenhuma verificação adicional que precisa ser feita em relação às informações da exceção lançada pelo método `mapear()`. Sendo assim, pode ser utilizado o parâmetro `expected` da anotação `@Test`. Nele, deve ser passada a classe da exceção que se espera que seja lançada. Dessa forma, o teste só irá passar caso a exceção seja lançada dentro do método de teste. Uma ressalva em relação ao uso do atributo `expected` é que o teste irá passar se a exceção for lançada em qualquer ponto do método de teste. Portanto, recomenda-se utilizar essa abordagem apenas no caso de a exceção poder ser lançada em apenas uma das chamadas de método do teste.

Finalizo essa seção ressaltando a importância da realização desse tipo de teste para classes que utilizam reflexão. É importante verificar como o componente está lidando com diferentes estruturas de classe, seja fazendo algum tratamento especial, ou lançando uma exceção informativa dizendo o motivo de aquela classe não ser aceita. Quando a classe precisar lidar com tipos, testes envolvendo tipos primitivos e arrays são sempre importantes de serem incluídos, pois eles na maioria das vezes requerem um tratamento especial. A criação de testes automatizados que abrangem diversos cenários diferentes é primordial para aumentar a qualidade da implementação, o que é especialmente importante se ela for reutilizada em diferentes contextos.

Para que complicar se o teste pode ser simples?

Os exemplos apresentados para os testes até o momento nesse livro possuem a estrutura mais básica possível para exercitar a funcionalidade desejada. Pode parecer que isso está sendo feito para manter as listagens pequenas ou para poder ser mais didáticos, porém na verdade é uma boa prática que os testes sejam simples e objetivos dentro da funcionalidade que está sendo exercitada. Dessa forma, por que criar um Java Bean com diversas propriedades, se duas já são o suficiente? O ideal é, a princípio, manter cada teste simples e focado em apenas uma funcionalidade, e posteriormente criar um teste mais complexo que envolve a execução de diversas funcionalidades combinadas.

5.2 Teste de proxies dinâmicos

Um outro tipo de teste importante quando trabalhamos com reflexão é o teste de proxies dinâmicos. A dificuldade de teste do proxy dinâmico está em sua própria natureza de encapsular um outro objeto. Sendo assim, uma importante parte do teste envolve a invocação de métodos nesse objeto encapsulado. Desse modo, dependendo da funcionalidade do proxy, é importante ver quais e quantas chamadas foram realizadas, quais parâmetros foram passados à classe encapsulada e mesmo simular diferentes cenários quando ela retornar diferentes valores e lançar exceções.

Vale ressaltar que, como o proxy é dinâmico, ele precisa estar preparado para lidar com diferentes tipos de classe. Portanto, devemos considerar que os métodos que vão ser recebidos pelo proxy podem possuir qualquer tipo de estrutura, com diferentes quantidades e tipos de parâmetros e diferentes tipos de retorno. Caso você decida não dar suporte a determinados tipos de método é importante verificar isso e lançar um erro se a interface possuir um método em um formato não aceito.

A estratégia básica para o teste de um proxy dinâmico é criar uma interface que capture uma característica que se queira testar e criar um mock object dessa interface. Em seguida, criar o proxy encapsulando o mock e utilizá-lo para as verificações relacionadas às invocações de método do proxy no objeto encapsulado.

Para exemplificar esse tipo de teste, essa seção apresenta testes da classe `CacheProxy` mostrada na seção 4.5. Nesse caso, o mock irá ajudar a verificar as situações em que o objeto encapsulado é invocado e os casos em que o resultado é obtido do cache armazenado pelo proxy. Para iniciar o teste, a listagem a seguir apresenta a interface `CacheMe`, que irá servir como a mock class para os testes criados.

Listagem 5.11 - Verificando informações das exceções recebidas:

```
public interface CacheMe{
    @Cache int metodoComCache(int param);
    @InvalidaCache void anulaCache();
}
```

O primeiro teste, implementado no método `cacheSimples()` da próxima listagem, verifica se o proxy armazena e retorna o valor do cache corretamente para chamadas de métodos com diferentes parâmetros. As expectativas do mock object é que `metodoComCache()` seja invocado uma vez com o parâmetro `1` e uma vez com o parâmetro `2`, sendo que ele vai retornar respectivamente os

valores `10` e `20`. Como ação do teste, o método `metodoComCache()` será invocado duas vezes com cada parâmetro e precisa receber os valores corretos como retorno. Dessa forma, será verificado se o método do mock object foi invocado somente na primeira vez, e se o cache armazenado foi retornado corretamente.

Listagem 5.12 - Verificando informações das exceções recebidas:

```
public class TesteCacheProxy {

    @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
    public void cacheSimples() {
        final CacheMe mock = ctx.mock(CacheMe.class);
        CacheMe proxy = (CacheMe) CacheProxy.criar(mock);

        ctx.checking(new Expectations() {{
            oneOf (mock).metodoComCache(1);
            will(returnValue(10));
            oneOf (mock).metodoComCache(2);
            will(returnValue(20));
        }});

        assertEquals(10, proxy.metodoComCache(1));
        assertEquals(20, proxy.metodoComCache(2));
        assertEquals(10, proxy.metodoComCache(1));
        assertEquals(20, proxy.metodoComCache(2));
    }
}
```

O segundo método de teste verifica a invocação de um método que anula o cache armazenado pelo proxy e está apresentado na listagem a seguir. Dessa vez, iremos invocar o método `metodoComCache()` duas vezes, e em seguida, o método `anulaCache()` e por fim, o método `metodoComCache()` mais duas vezes. A expectativa é que `metodoComCache(0)` seja invocado apenas duas vezes no mock object, retornando o valor `10` na primeira vez e o valor `20` da segunda vez, isso além da chamada do método `anulaCache()`. Observe que existem asserções que verificam se as duas primeiras chamadas retornam `10` e se as duas últimas retornam `20`. Esse teste verifica se o cache é limpo depois da chamada de um método anotado com `@InvalidaCache`.

Listagem 5.13 - Verificando informações das exceções recebidas:

```
@Test
public void invalidaCache() {
    final CacheMe mock = ctx.mock(CacheMe.class);
    CacheMe proxy = (CacheMe) CacheProxy.criar(mock);

    ctx.checking(new Expectations() {{
        exactly(2).of(mock).metodoComCache(1);
        will(onConsecutiveCalls(returnValue(10),returnValue(20)));
        oneOf(mock).anulaCache();
    }});

    assertEquals(10, proxy.metodoComCache(1));
    assertEquals(10, proxy.metodoComCache(1));
    proxy.anulaCache();
    assertEquals(20, proxy.metodoComCache(1));
    assertEquals(20, proxy.metodoComCache(1));
}
```

É importante ficar claro que esses dois testes apenas ilustram a abordagem para o teste de um proxy dinâmico. Em uma suite de testes mais completa para esse proxy, seriam testados cenários mais complexos, como uma interface que possui métodos com o mesmo nome, métodos com vários parâmetros, métodos com quantidade variável de parâmetros, retorno e parâmetros com objetos complexos, entre outros. Por exemplo, imagine dois métodos com mesmo nome que recebem respectivamente um e dois inteiros como parâmetro, será que o proxy usaria o valor do retorno da chamada com parâmetros `(1,1)` para a chamada com `(11)`? Deixo como exercício para o leitor implementar esse teste, e corrigir a implementação do proxy se necessário.

5.3 Testando a configuração de metadados

As seções anteriores desse capítulo abordaram a criação de testes que verificam os componentes ou classes que utilizavam reflexão e metadados. Porém, quem garante que esse componente vai se comportar conforme o desejado para um determinada classe? Por exemplo, se você quer que os parâmetros da sua aplicação sejam mapeados para uma classe, é preciso saber se isso está sendo feito da forma correta. Da

mesma forma, se você deseja que o cache seja feito para uma determinada classe utilizando a classe `CacheProxy`, é preciso verificar se para a estrutura criada para a classe e as anotações configuradas refletem esse comportamento.

Quando estamos utilizando um componente que utiliza reflexão e metadados, a própria estrutura da classe e os metadados vão determinar o seu comportamento quando essa classe ou um de seus objetos forem passados como parâmetro. Por mais que pareça estranho testarmos algo declarativo, é importante saber se os metadados estão declarados de forma a gerar o comportamento adequado. Enquanto os testes da classe ou do componente são feitos pelos seus desenvolvedores, esse tipo de teste dos metadados deve ser feito por quem o está utilizando.

Como testar o comportamento gerado pelos metadados

Para criarmos um teste que verifique se os metadados estão configurados corretamente, deve ser checado o comportamento da classe que utiliza reflexão ao receber uma determinada classe da aplicação. O teste é bem parecido com os que foram mostrados nas seções anteriores, com a diferença que não será utilizada uma mock class e sim uma classe real da aplicação. O objetivo também é diferente, pois não queremos verificar se a classe que utiliza reflexão funciona corretamente quando recebe uma classe com uma determinada estrutura e anotações; mas sim, assumindo que o componente foi testado e funciona corretamente, deseja-se verificar quais são a estrutura e anotações que a classe precisaria ter para se obter um determinado comportamento.

Imagine que em minha aplicação, eu precise receber como parâmetros na linha de comando uma lista de arquivos de entrada e apenas um arquivo de saída. Para facilitar a implementação da leitura desses parâmetros, será utilizado o componente de mapeamento de parâmetros apresentado na seção 3.5. Sendo assim, o teste envolverá invocar a classe `MapeamentoParametros` com a classe da minha aplicação e ver se ela recebe as informações da forma correta. A listagem a seguir mostra o teste que poderia ser feito para essa verificação.

Listagem 5.14 - Verificando se o mapeamento foi feito de forma correta:

```
@Test
public void arquivosEntradaSaida() {
    String[] a =
        {"-in","entradaA.txt","entradaB.txt","-out","saida.txt"};
    MapeamentoParametros<ParametrosArquivo> map =
```

```
            new MapeamentoParametros<>(ParametrosArquivo.class);
    ParametrosArquivo p = map.mapear(a);

    assertEquals("entradaA.txt", p.getArquivosEntrada()[0]);
    assertEquals("entradaB.txt", p.getArquivosEntrada()[1]);
    assertEquals("saida.txt", p.getArquivoSaida());
}
```

Na posição de quem está implementando a aplicação, é desejável que seja lançada uma exceção quando forem passados mais de um arquivo de saída. Dessa forma, mais um método de teste é adicionado para verificar esse cenário. A listagem a seguir mostra a verificação para saber se uma exceção é lançada no caso de dois arquivos de saída serem passados como parâmetro.

Listagem 5.15 - Verificando se o componente lança exceção com parâmetros errados:

```
@Test(expected=MapeamentoException.class)
public void maisDeUmArquivoDeSaida() {
    String[] a =
        {"-in","entradaA.txt","-out","saidaA.txt", "saidaB.txt"};
    MapeamentoParametros<ParametrosArquivo> map =
            new MapeamentoParametros<>(ParametrosArquivo.class);
    ParametrosArquivo p = map.mapear(a);
}
```

Uma questão importante nesse tipo de teste é que a verificação é feita de acordo com os requisitos da aplicação. Sendo assim, o comportamento desejado pode ser obtido através de uma combinação entre metadados e o comportamento do componente que utiliza reflexão. Por exemplo, imagine que essa aplicação possua um parâmetro opcional para o timeout do processamento, que é do tipo inteiro, porém o componente só aceita parâmetros do tipo `String`, array de `String` e booleano. A listagem a seguir mostra como seria o teste e a implementação para esse exemplo.

Listagem 5.16 - Verificando se o componente lança exceção com parâmetros errados:

```
@Test
public void parametroTimeout() {
```

```
    String[] a = {"-time","5000"};
    MapeamentoParametros<ParametrosArquivo> map =
            new MapeamentoParametros<>(ParametrosArquivo.class);
    ParametrosArquivo p = map.mapear(a);
    assertEquals(5000, p.getTimeout());
}

//implementação
public class ParametrosArquivo {

    @Parametro("-time")
    private int timeout;

    public void setTimeout(String timeout) {
        this.timeout = Integer.parseInt(timeout);
    }
}
```

Observe que o teste expressa somente o comportamento esperado, não se importando se a funcionalidade será implementada somente pelo componente ou se vai precisar ser complementada nas classes da aplicação. No caso, é esperado que seja retornado um valor inteiro pelo método de acesso na classe `ParametrosArquivo`. Sendo assim, para gerar esse resultado esperado, como mostrado na listagem, essa classe precisaria fazer a conversão do valor no momento que recebesse a `String` da classe `MapeamentoParametros`.

Como reutilizar código de teste

Quando utilizamos componentes e classes baseados em reflexão e metadados, é possível reutilizá-lo para executar funcionalidade para diversas classes da aplicação. Apesar desse reúso poupar tempo de desenvolvimento, os testes do funcionamento do componente para cada classe da aplicação pode ser trabalhoso e tomar muito tempo. Porém, se avaliarmos o teste do mesmo componente para diferentes classes, é possível percebem que eles possuem muito em comum, como a chamada do próprio método do componente. Nessa seção, vamos ver como é possível reutilizar código desse tipo de teste, para que se possa poupar tempo de desenvolvimento também nessa etapa.

A listagem a seguir mostra o exemplo de um teste que verifica a geração do mapa para a classe `Produto`. Nesse caso, o atributo `descricao` precisa ser ignorado

e o atributo `categoria` deve ter ser inserido no mapa como `"tipo"`. Observe que as asserções verificam as informações que estão no mapa e os atributos que não devem estar no mapa. Como já foi dito, é importante lembrar que o que está sendo testado não é o `GeradorMapa`, mas os metadados configurados.

Listagem 5.17 - Testando a geração do mapa para a classe Produto:

```
public class TesteMapaProduto {

    @Test
    public void geracaoMapa() {
        Produto p = new Produto();
        p.setNome("Tablet APX10");
        p.setPreco(500.00);
        p.setDescricao("10 pol, 16GB, 800x600");
        p.setCategoria("Eletrônicos");

        Map<String,Object> mapa = GeradorMapa.gerarMapa(p);

        assertEquals(mapa.get("nome"), "Tablet APX10");
        assertEquals(mapa.get("preco"), 500.00);
        assertEquals(mapa.get("tipo"), "Eletrônicos");
        assertFalse(mapa.containsKey("descricao"));
        assertFalse(mapa.containsKey("categoria"));
    }
}
```

Para ser possível reutilizar a classe de teste, o primeiro passo é separar a parte do teste que é mais geral do que a que é específica para a classe `Produto`. A ideia é que métodos auxiliares forneçam as informações específicas para serem utilizadas em um método de teste genérico. Na listagem a seguir, isso foi feito para a classe de teste mostrada anteriormente. O método `getObjeto()` cria e popula o objeto que será utilizado para o teste, e os métodos `getConteudoMapa()` e `getExcluidos()` retornam respectivamente as entradas esperadas para o mapa e as chaves que não devem estar presentes no mapa.

Listagem 5.18 - Separando a parte geral do teste da parte específica para a classe Produto:

```
public class TesteMapaProduto {
```

```
    @Test
    public void geracaoMapa() {
        Object o = getObjeto();
        Map<String,Object> mapa = GeradorMapa.gerarMapa(o);
        Map<String,Object> esperado = new HashMap<>();
        getConteudoMapa(esperado);
        for(String s : esperado.keySet()){
            assertEquals(esperado.get(s), mapa.get(s));
        }
        List<String> excluidos = new ArrayList<>();
        for(String s : excluidos){
            assertFalse(mapa.containsKey(s));
        }
    }
    protected Object getObjeto() {
        Produto p = new Produto();
        p.setNome("Tablet APX10");
        p.setPreco(500.00);
        p.setDescricao("10 pol, 16GB, 800x600");
        p.setCategoria("Eletrônicos");
        return p;
    }
    protected void getConteudoMapa(Map<String, Object> esperado){
        esperado.put("nome", "Tablet APX10");
        esperado.put("preco", 500.00);
        esperado.put("tipo", "Eletrônicos");
    }
    protected void getExcluidos(List<String> excluidos){
        excluidos.add("descricao");
        excluidos.add("categoria");
    }
}
```

A seguir, como mostrado na próxima listagem, pode-se extrair uma superclasse abstrata que contém a implementação do método de teste e a declaração abstrata dos métodos mais específicos. Dessa forma, a classe `TesteMapaProduto` apenas implementa esses métodos auxiliares, provendo informações para o teste que foi definido na superclasse. Como a chamada de método é a mesma, é possível fazer isso, pois o que muda é apenas o objeto passado para o método e o que é esperado para o retorno.

Listagem 5.19 - Criando uma classe modelo para a criação de testes do gerador de mapas:

```
public abstract class TesteMapaGenerico {

    @Test
    public void geracaoMapa() {
        Object o = getObjeto();
        Map<String,Object> mapa = GeradorMapa.gerarMapa(o);
        Map<String,Object> esperado = new HashMap<>();
        getConteudoMapa(esperado);
        for(String s : esperado.keySet()){
            assertEquals(esperado.get(s), mapa.get(s));
        }
        List<String> excluidos = new ArrayList<>();
        for(String s : excluidos){
            assertFalse(mapa.containsKey(s));
        }
    }
    protected abstract Object getObjeto();
    protected abstract void getConteudoMapa(Map<String,
                                            Object> esperado);
    protected abstract void getExcluidos(List<String> excluidos);
}
```

Por mais que a quantidade de linhas de código não mude muito, ter apenas métodos simples para implementar certamente aumenta a velocidade de criação dos testes de outros objetos. Esse efeito se potencializa, quando existem outros métodos e outros cenários a serem testados.

5.4 Gerando classes com ClassMock

Uma das dificuldades de testar classes e componentes que utilizam reflexão é o fato de precisar a cada teste definir uma classe para ser passada como parâmetro. Algumas vezes, a diferença entre classes definidas em diferentes testes é apenas a assinatura de um método ou a presença de uma anotação. Infelizmente, não dá para reutilizar a definição de uma classe para a outra, por mais que elas sejam muito similares. Isso acaba tornando a definição de teste para esse tipo de classe mais trabalhosa e mais verbosa, pois as classes precisam ser definidas sempre por inteiro.

Visando simplificar esse tipo de teste, o framework **ClassMock** [1] gera classes em tempo de execução para serem passadas como parâmetro. Esse framework pode ser baixado no endereço http://classmock.sf.net. Ao invés de criar uma classe da forma usual, como foi visto nas seções anteriores, a partir da classe `ClassMock` são inseridas as informações da classe que se deseja criar, como propriedades (que incluem atributo e métodos de acesso), interfaces, superclasse, métodos, anotações, entre outros. Dessa forma, é possível reutilizar uma definição inicial como base para gerar as variações necessárias para um método, ou mesmo criar métodos que insiram determinadas estruturas em uma mock class. O objetivo dessa seção é apresentar a abordagem utilizada pelo ClassMock, sem necessariamente apresentar todos os métodos disponíveis em sua API. Para um conhecimento mais aprofundado de todas as funcionalidades dessa ferramenta, sugere-se acessar a documentação disponível no site do projeto.

Utilizando o ClassMock no gerador de mapas

Para exemplificar o uso do ClassMock, a listagem a seguir mostra os mesmos testes desenvolvidos para a classe `GeradorMapa` na seção 5.1, criados agora utilizando esse framework. Observe que o método `criarMockClass()` é anotado com `@Before` e por isso será executado antes de todos os métodos de teste. Nesse método, é criado uma classe base para os testes chamada de `Bean` e que possui as propriedades `prop1` do tipo `String` e `prop2` inteira, da mesma forma que no exemplo original. Um atributo da classe de teste chamado `mockClass` do tipo `ClassMock` é utilizado para guardar essas informações.

Listagem 5.20 - Testes do gerador de mapas utilizando o ClassMock:

```
public class TesteGeradorMapaClassMock {

    private ClassMock mockClass;

    @Before public void criarMockClass(){
        mockClass = new ClassMock("Bean");
        mockClass.addProperty("prop1", String.class)
        .addProperty("prop2", int.class);
    }
    @Test public void mapaDeClasseSimples() {
        Object instance = ClassMockUtils.newInstance(mockClass);
        ClassMockUtils.set(instance,"prop1","teste");
```

```
        ClassMockUtils.set(instance,"prop2",25);
        Map<String,Object> mapa = GeradorMapa.gerarMapa(instance);
        assertEquals("teste",mapa.get("prop1"));
        assertEquals(25,mapa.get("prop2"));
    }
    @Test public void ignoraPropriedade() {
        mockClass.addAnnotation(
            "prop1", Ignorar.class, Location.GETTER);
        Object instance = ClassMockUtils.newInstance(mockClass);
        ClassMockUtils.set(instance,"prop1","teste");
        ClassMockUtils.set(instance,"prop2",25);
        Map<String,Object> mapa = GeradorMapa.gerarMapa(instance);
        assertFalse(mapa.containsKey("prop1"));
        assertEquals(25,mapa.get("prop2"));
    }
    @Test public void mudaNomePropriedade() {
        mockClass.addAnnotation("prop2", NomePropriedade.class,
            Location.GETTER, "propriedade2");
        Object instance = ClassMockUtils.newInstance(mockClass);
        ClassMockUtils.set(instance,"prop1","teste");
        ClassMockUtils.set(instance,"prop2",25);
        Map<String,Object> mapa = GeradorMapa.gerarMapa(instance);
        assertEquals("teste",mapa.get("prop1"));
        assertEquals(25,mapa.get("propriedade2"));
        assertFalse(mapa.containsKey("prop2"));
    }
}
```

Em seguida, em cada método de teste é criado um novo objeto a partir da mock class definida, no qual são configuradas propriedades. Como a classe é definida em tempo de execução, é preciso utilizar reflexão para manipular seus métodos e a instanciar, porém o próprio ClassMock possui uma classe chamada `ClassMockUtils` com métodos auxiliares que podem ser utilizados para isso. No exemplo, são usados os métodos `newInstance()`, que retorna uma instância de uma classe definida com `ClassMock`, e `set()`, que insere um valor em uma propriedade. Outros métodos dessa classe são `get()`, que recupera informações, e `invoke()`, que executa métodos. Caso o método testado não receba uma instância da classe, mas uma instância de `Class` representando a classe, ela pode ser recuperada a partir do método `createClass()` da própria classe `ClassMock`.

Observe no exemplo de classe de teste mostrada que os métodos de teste

ignoraPropriedade() e mudaNomePropriedadea() adicionam uma anotação na mock class antes de criarem uma instância dela. Dessa forma, é possível aproveitar a estrutura inicial da classe definida no método anotado com @Before para depois fazer as modificações necessárias nos métodos de teste. Peço ao leitor para voltar na seção 5.1 e ver como dessa forma é possível definir os testes de forma mais objetiva e menos verbosa.

ClassMock + JMock = testes de proxies dinâmicos

A ideia principal do ClassMock é criar a estrutura de mock classes para serem utilizadas para teste. Quando são adicionadas propriedades, o ClassMock gera não só o atributo, mas também a implementação dos dois métodos de acesso. Porém, não é o seu objetivo gerar o comportamento de métodos, mas somente a sua estrutura. Quando é necessário simular algum comportamento, é preciso utilizar algum framework de mock objects, como o JMock, para criar o mock object da mock class definida. Esse tipo de prática, como foi visto na seção 5.2, é o caminho que deve ser utilizado para a criação de teste para proxies dinâmicos.

A listagem a seguir mostra como essa abordagem seria utilizada combinando o uso dos frameworks ClassMock e JMock para o teste da classe CacheProxy. Inicialmente, a classe ClassMock é utilizada para criar uma interface chamada CacheMe. O segundo parâmetro true indica que uma interface será criada. Em seguida, é adicionado um método abstrato chamado comCache() que retorna inteiro, indicado pelo primeiro parâmetro, e recebe um inteiro, indicado pelo último parâmetro. Por fim, é adicionada a anotação @Cache no método comCache(). Todos esses parâmetros serão utilizados para a geração dinâmica da interface para ser utilizada pelo teste, que é feita no momento que o método createClass() é chamado.

Listagem 5.21 - Testes do gerador de mapas utilizando o ClassMock:

```
public class TesteCacheProxyClassMock {

    @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

    @Test
    public void cacheSimples() throws Throwable {
        ClassMock cm = new ClassMock("CacheMe", true);
        cm.addAbstractMethod(int.class, "comCache", int.class)
```

```
            .addMethodAnnotation("comCache", Cache.class);
        Class<?> interf = cm.createClass();
        final Object mock = ctx.mock(interf);
        Object proxy = CacheProxy.criar(mock);

        ctx.checking(new Expectations() {{
            ClassMockUtils.invoke(one(mock),"comCache",1);
            will(returnValue(10));
        }});

        assertEquals(10, ClassMockUtils.invoke(proxy,"comCache",1));
        assertEquals(10, ClassMockUtils.invoke(proxy,"comCache",1));
    }
}
```

O próximo passo depois da criação da mock class, é a criação do respectivo mock object para definir as expectativas e o comportamento do objeto que será encapsulado pelo proxy dinâmico. O método `mock()` é chamado recebendo a interface criada dinamicamente, e retorna uma instância dela controlada pelo JMock. Em seguida, o proxy é criado a partir do encapsulamento do mock object. Depois disso, a lógica do teste é a mesma, porém, como a interface foi criada de forma dinâmica, é preciso utilizar o método `invoke()` da classe `ClassMockUtils` para fazer a invocação dos métodos. Note que esse método é utilizado nas asserções e inclusive para definir as expectativas do mock object.

5.5 Considerações finais

Esse capítulo apresentou as técnicas básicas para o teste de classes que utilizam reflexão e metadados para seu processamento. Inicialmente foi mostrada a criação de classes para simular diferentes estruturas que serão consumidas pelos testes, e em seguida como as invocações de métodos poderiam ser verificadas. Baseado nos exemplos apresentados, foi introduzido o conceito de **mock class** que para o teste desse tipo de classe é diferente e complementar aos **mock objects**. Também foi abordado o teste de proxies dinâmicos, que combinam os conceitos de mock class e mock object para a definição do teste. Por fim, vimos o teste da configuração de metadados, que deve ser feito pelas aplicações que utilizam esses componentes para verificar se os metadados estão configurados corretamente para que o comportamento seja o esperado.

Em geral, o teste desse tipo de componente é difícil de ser realizado pois depende de fatores dinâmicos nas classes, que muitas vezes são difíceis de serem previstos e simulados. Porém, essa dificuldade é um obstáculo que não pode impedir a criação desses testes, pois, por ser reutilizado em diversas partes do sistema, é preciso que ele possua uma boa qualidade. Frameworks como o JMock ou o ClassMock ajudam a diminuir algumas dificuldades inerentes a esse tipo de teste, porém dependendo da funcionalidade da classe, novas dificuldades podem surgir, e podem demandar até a criação de outras classes auxiliares para viabilizar a criação do teste.

Referências Bibliográficas

[1] Clovis Fernandes Eduardo Guerra, Fabio Silveira. Classmock: A testing tool for reflective classes which consume code annotations. WorkshopBrasileirodeMétodosÁgeis(WBMA2010), 2010.

[2] Hans Rohnert Peter Sommerlad Michael Stal Frank Buschmann, Regine Meunier. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

[3] Pattie Maes. Consepts and experiments in computational reflection. ProceedingsoftheInternationalConferenceonObjectOrientedProgramming, Systens,LanguagesandApplications, 1987.

[4] Java Community Process. Javabeans(tm) specification 1.01 final release. http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/, 1997.

[5] Brian Smith. Reflection and semantics in a procedural language. TesedeDoutorado-DepartmentElectricalandComputerScienceEngineering, MassachusettsInstituteofTechnology, 1982.